VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Implementace algoritmu Finite State Entropy
Finite State Entropy Implementation

2019 Bc. Daniel Hrtús

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Daniel Hrtús**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Implementace algoritmu Finite State Entropy
Finite State Entropy Implementation

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je popsat a implementovat algoritmus Finite State Entropy pro kompresi dat. Algoritmus vychází z prací Jarka Dudy. Cílem implementace je funkční program, se kterým bude možno experimentovat a optimalizovat jej pro různé potřeby.

Práce bude obsahovat:
1. Rozbor algoritmu Finite State Entropy a Asymetric Numeral Systems.
2. Návrh implementace algoritmu.
3. Implementace algoritmu.
4. Otestování algoritmu nad zvolenými daty.
5. Porovnání algoritmu s jinými přístupy (Huffmanovo kódování, Aritmetické kódování).

Seznam doporučené odborné literatury:

[1] Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding, Jarek Duda, 2013, http://arxiv.org/abs/1311.2540v2
[2] Handbook of Data Compression, Salomon D., Motta G., Springer; 5th edition, 2010

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **doc. Ing. Jan Platoš, Ph.D.**

Datum zadání: 01.09.2017

Datum odevzdání: 30.04.2019

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 28. 6. 2019 .....

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.

V Ostravě 28. 6. 2019 .....

Chtěl bych poděkovat panu doc. Ing. Janu Platošovi, Ph.D. za odborné vedení, trpělivost a ochotu, kterou mi v průběhu zpracování diplomové práce věnoval.

Abstrakt

V posledních letech rapidně přibývá objem elektronických dat. Nezvětšuje se pouze jejich velikost, ale také počet. Pro lepší manipulaci, jejich přenos a uchování, je nutné velikost těchto dat co nejvíce zmenšit pomocí komprese. Jednou metodou komprese dat je entropické kódování, které je univerzální pro jakýkoliv typ dat a v mnoha případech je součástí kompresních programů. V této práci jsou nejprve stručně představeny současně nejpoužívanější metody entropického kódování. Poté bude přiblížena relativně nová metoda s názvem Asymmetric Numeral Systems (ANS). Hlavní část je zaměřena na implementaci jedné z variant ANS, která je v této práci podrobněji popsána. Na závěr je tato implementace otestována pro různá vstupní data a zhodnocena její efektivita.

Klíčová slova

komprese dat, ANS, entropické kódování, FSE, Huffmanovo kódování, aritmetické kódování

Abstract

The volume of electronic data is growing rapidly in the last years. Not only grows the number of records but also their size. For better manipulation, especially during transmission and archiving, there is a need to reduce the size as much as possible using data compression. Entropy coding, which is a compression method that is independent of input data, is generally used in compression programs. At the beginning of the thesis, there are briefly summarized currently most used entropic coding methods. After that, the thesis continues with a description of a new method called Asymmetric Numeral Systems (ANS). The main part focuses on implementing one of the variants of ANS. In the end, the implementation is tested on different data inputs to review its effectivity.

Keywords

data compression, ANS, entropy coding, FSE, Huffman coding, arithmetic coding

Obsah

Seznam použitých zkratek a symbolů

ANS – Asymmetric Numeral Systems

FSE – Finite State Entropy

LZ – Lempel-Ziv

AAC – Advanced Audio Coding

LZW – Lempel-Ziv-Welch

LZSS – Lempel-Ziv-Storer-Szymanski

ABS – Asymmetric Binary System

Seznam tabulek

Seznam obrázků

Seznam kódů

1 Úvod

Žijeme v době, kdy je produkováno enormní množství elektronických dat. Ať už to jsou e-maily, data ze senzorů, elektronické dokumenty nebo například statistická data. S těmito daty se obvykle musí dále manipulovat, jelikož je nutné je přenášet nebo archivovat. Pokud jsou přenášeny velké objemy dat, jejich přenos může trvat velmi dlouhou dobu, v horším případě vůbec nemusí stačit kapacita přenosové linky a může dojít k selhání. Pro archivaci je pak nutné dostatečně velké datové úložiště. Naléhavost těchto požadavků roste úměrně s počtem dat a jejich objemem. To však neplatí o přenosové rychlosti či kapacitě úložišť.

Jedním ze způsobů, kterým lze tyto nároky snížit je komprese dat. Tím dojde ke zmenšení velikosti dat na přijatelnější hodnoty pro další zpracování. Komprese dat je dnes zásadním postupem prakticky ve všech moderních elektronických systémech. Ve 20. století byla objevena velká řada kompresních metod, obvykle však platí, že ne všechny lze použít pro jakýkoliv typ dat. Mezi specifické druhy komprese patří například slovníkové metody, jejichž nejznámějším zástupcem je metoda LZ77 a její modifikace, které jsou vhodné zejména pro textová data. Dále existují metody, které jsou obecné a lze je použít pro jakýkoliv typ dat s přijatelnými výsledky. Tyto metody se obecně označují pojmem entropické kódování. Mezi nejznámější zástupce patří Huffmanovo a aritmetické kódování. Zjednodušeně lze říci, že Huffmanovo kódování je rychlé, ale méně účinné, zatímco aritmetické kódování je pomalé, ovšem s téměř ideálními výsledky. Obě tyto metody byly vynalezeny před více než 30 lety a až do roku 2009, kdy byla vynalezena nová metoda entropického kódování s názvem Asymmetric Numeral Systems (ANS), nedošlo v oblasti komprese dat k téměř žádnému pokroku.

Metoda ANS slibuje dosažení optimálních výsledků porovnatelných s aritmetickým kódováním, a to za dobu srovnatelnou s Huffmanovým kódováním. V minulosti se při rozhodování o použití konkrétního entropického kódování muselo počítat s některými omezeními. A to buď s rychlostí nebo účinností komprese. Dle pozorovaných výsledků se však již s těmito nevýhodami u metody ANS nemusíme potýkat.

Cílem práce je implementace algoritmu Finite State Entropy (FSE), který je konkrétní implementací jedné z variant metody ANS. Práce je rozdělena na čtyři hlavní části. V první je představena komprese dat spolu s možnými rozděleními na určité typy. Každý z těchto typů má jiné využití a hodí se pro různá data. Také jsou v této kapitole přiblíženy zmíněné dvě metody entropického kódování, tedy Huffmanovo a aritmetické kódování. Třetí kapitola se zabývá novou metodou ANS, kde je podrobněji popsán její základní princip a také možné varianty. Kapitola dále pokračuje popisem jedné zvolené varianty (tANS). V kapitole čtvrté je pak proveden rozbor implementace této metody, který zahrnuje ukázky nejdůležitějších funkcí a částí tohoto programu. Pátá kapitola je zaměřena na testování vzniklého programu, pro data různého typu. Bylo provedeno několik různých testů a porovnání s ostatními metodami. Závěrem této práce je také vlastní implementace, její zhodnocení z hlediska efektivity a odhady budoucích možností.

2 Komprese dat

V informatice se kompresí dat rozumí odstranění nadbytečných nebo duplicitních dat z informace. Hlavním důvodem je typicky zmenšení velikosti informace pro další zpracování jako je třeba přenos nebo archivace. Opakem komprese je dekomprese a tento postup slouží k zpětnému získání původní informace. Komprese dat se dá rozdělit na dva hlavní typy:

- Ztrátová
- Bezztrátová

Při použití bezztrátové komprese nedochází ke ztrátě informace. To znamená, že informace po dekompresi je naprosto totožná jako informace, která byla použita před kompresí. Naopak pokud danou informaci zkomprimujeme ztrátovým algoritmem, již nelze získat originální informaci zpět ve stejném tvaru jako před kompresí. Obvykle se ztrátová komprese využívá tam, kde lze připustit ztrátu informace, typicky tedy obraz a zvuk. Známými zástupci z obrazových metod jsou *JPEG* pro statické obrázky nebo například *H.264*, který je určen pro video. Pro zvuk jsou známé metody *MP3* nebo *AAC*. Bezztrátovou kompresi využijeme všude, kde je potřebné zachovat originální informaci. Nejčastější využití nalezne u textových souborů, jedním z nejznámějších zástupců je metoda *DEFLATE*, která je základ *ZIP* souborů. Ovšem využití bezztrátové komprese není limitováno typem souboru. Existují také bezztrátové kompresní algoritmy pro zvukové, obrazové či video soubory.

Bezztrátovou kompresi můžeme nadále členit na dvě skupiny:

- Slovníkové metody
- Entropické kódování

Slovníkové metody jsou efektivní zejména v textových souborech, jelikož jsou založeny na hledání nejdelší shody v datech. Mezi nejznámější zástupce patří metody z rodiny *LZ*, jako například *LZ77*, *LZ78*, *LZW*, *LZSS* a další varianty. Původní variantu *LZ77* vynalezli Abraham Lempel a Jacob Ziv v roce 1977. V dnešní době jsou tyto metody používány spolu s entropickým kódováním pro dosažení co nejlepších výsledků komprese a jsou součástí populárních kompresních programů jako jsou *ZIP*, *7-Zip*, *RAR* a jiných.

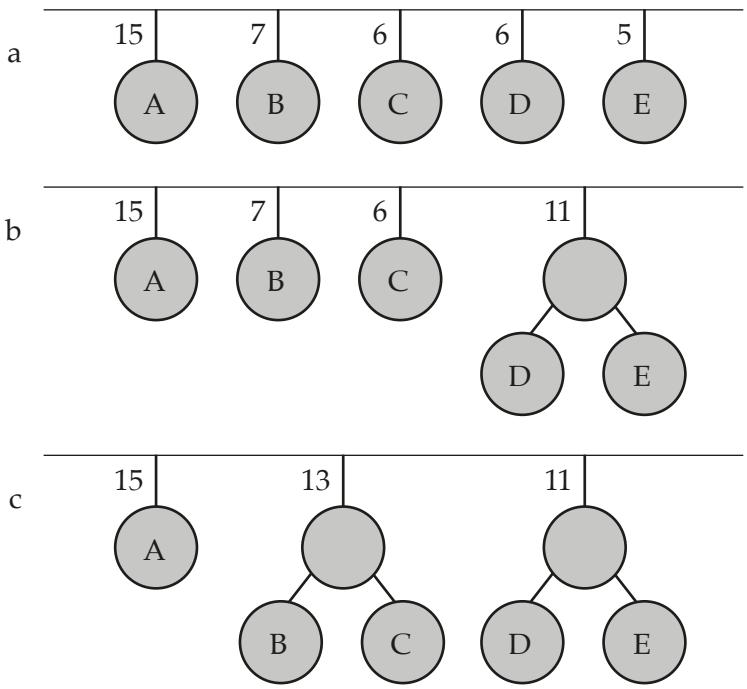
Princip entropického kódování je nahrazení vstupních symbolů sekvencí kódů proměnné délky za účelem zkrácení originálního vstupu. Symboly s největší pravděpodobností výskytu budou reprezentovány nejkratšími kódy a symboly s nižší pravděpodobností kódy delšími. Například pokud je pravděpodobnost výskytu symbolu 50 %, ideální je použít 1 bit pro kódování, pokud má 25 %, použili bychom 4 bity. Tato hodnota je dána vztahem $-log_2(P)$. Americký matematik Claude Elwood Shannon v roce 1948 jeho prací *A Mathematical Theory of Communication* [1] dokázal, že pod tuto hodnotu se bezztrátovou kompresí nedá dostat, ale můžeme se jí pouze přiblížit. Shannon mimo jiné také definoval entropii (v teorii informací) jako míru informací obsažené ve zprávě. Entropie je definována takto:

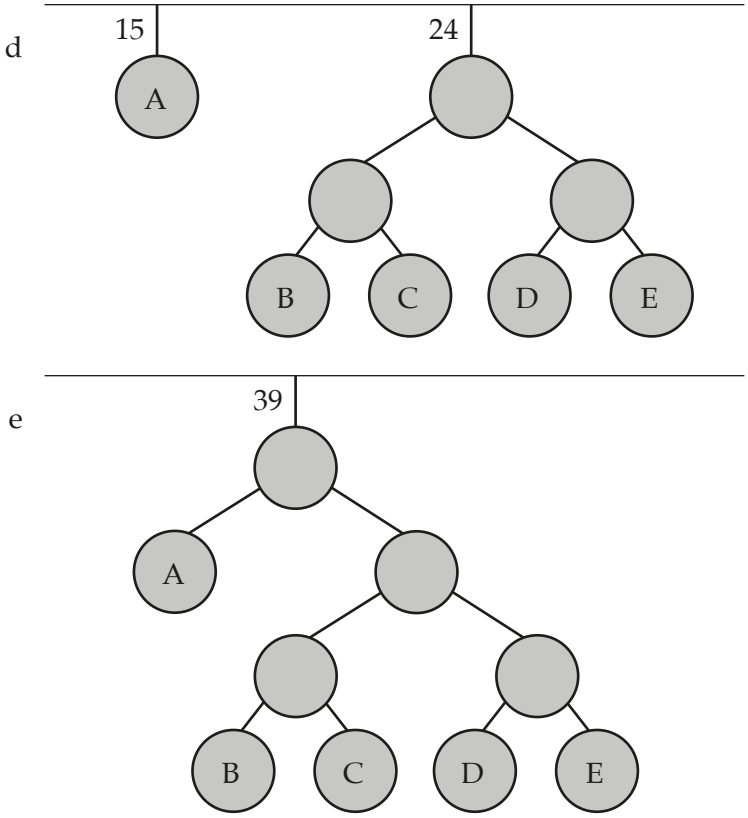
$$H(X) = -\sum_{i=1}^{n} P(x_i)\, log_b\, P(x_i)$$

Od zjištění tohoto faktu se lidé snažili objevit metodu, která by generovala kódy velmi blízko tomuto limitu. První řešení, které bylo dlouhou dobu považováno za optimální, vymyslel David Albert Huffman roku 1952. [2] Nicméně Huffmanovo kódování mělo určitá omezení. O několik let později Jorma Rissanen představil novou metodu – aritmetické kódování. Od této doby výzkum entropického kódovaní stagnoval. Lepších výsledků se povedlo dosáhnout až po roce 2009, kdy Jarosław Duda poprvé představil metodu jménem Asymmetric numeral systems (ANS). [3]

2.1 Huffmanovo kódování

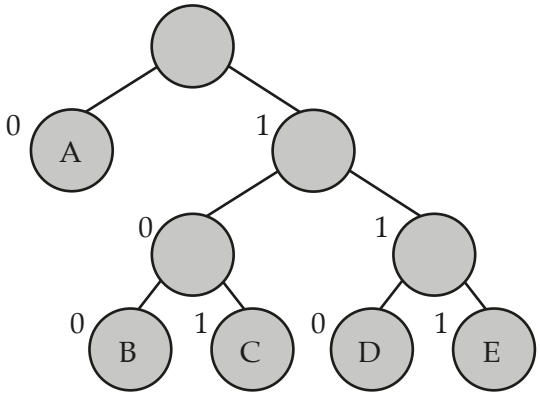
Problém s entropickým kódováním poprvé velmi úspěšně vyřešil David A. Huffman. Jeho metoda je založena na generování kódových slov proměnné délky v závislosti na pravděpodobnosti výskytu symbolu. Jedná se o takzvaný prefixový kód, což znamená, že žádný kód není prefixem jakéhokoliv jiného kódu. To umožní přesně určit, který symbol dané kódové slovo označuje. Huffmanovo kódování vytváří binární strom, který obsahuje jednotlivé symboly. Z počátku jsou všechny uzly stromu listovými a postupně se strom upravuje. Nejprve se všechny uzly označí váhou, která je závislá na pravděpodobnosti výskytu symbolu – častější symboly mají větší váhu a naopak. Poté se provádí spojování uzlů, dokud není pouze jeden kořen. V každém kroku se spojí dva uzly s nejmenší váhou. Spojení probíhá vytvořením jednoho společného uzlu pro tyto dva listové uzly. Výsledkem je binární strom, kde všechny listové uzly obsahují jeden konkrétní symbol. Příklad tvorby jednoduchého stromu ilustruje obrázek 1. [4]





Obrázek 1: Tvorba Huffmanova stromu

Poté se všechny uzly ve stromu označí hodnotou 0 nebo 1, která bude reprezentovat po průchodu stromem kódové slovo pro daný symbol. Při kódování se prochází stromem od listu ke kořenu. Symbol B tedy zakódujeme jako 100, D jako 110 a tak dále. Dekódování není potom nic jiného než pouhé procházení stromem od kořenu, dokud nenarazíme na list. Procházení stromu opakujeme tak dlouho, dokud jsou na vstupu nějaké bity. Ukázka Huffmanova stromu je na obrázku 2.

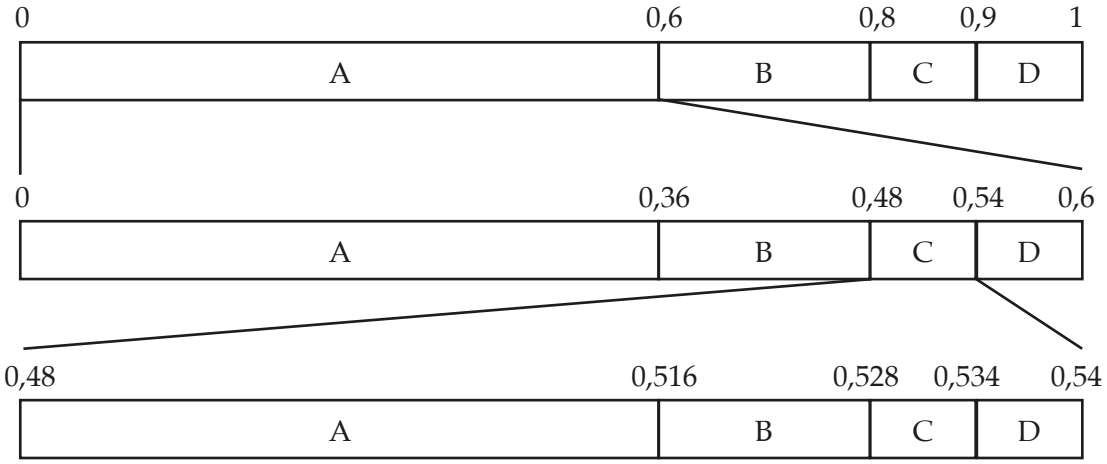


Obrázek 2: Huffmanův strom

Huffmanovo kódování je rychlé především proto, že jak při kompresi tak dekompresi nejsou potřeba složitější instrukce jako násobení a dělení. Také není náročné na operační paměť, a díky těmto dvěma vlastnostem je vhodné pro použití v zařízeních, kde je operační paměť limitujícím faktorem nebo za účelem úspory energie. Nicméně Huffmanovo kódování má také jednu zásadní nevýhodu, a to že nikdy nepokoří hranici minimálně jednoho bitu na symbol. Například symbol s pravděpodobností výskytu 0,85 by měl být zakódován ideálně pomocí 0,23 bitů. To není u Huffmanova kódování možné, jelikož minimální délka kódu bude vždy 1. Není to ovšem problém pouze symbolů s pravděpodobností nad 50 %, také symbol s pravděpodobností 18 % má být ideálně zakódován pomocí 2,47 bitů. Dosáhnout optimálních kódů je s Huffmanovým kódováním možné pouze v případě, že pravděpodobnosti výskytu symbolů jsou mocniny čísla 2. To ovšem v reálných případech nelze očekávat.

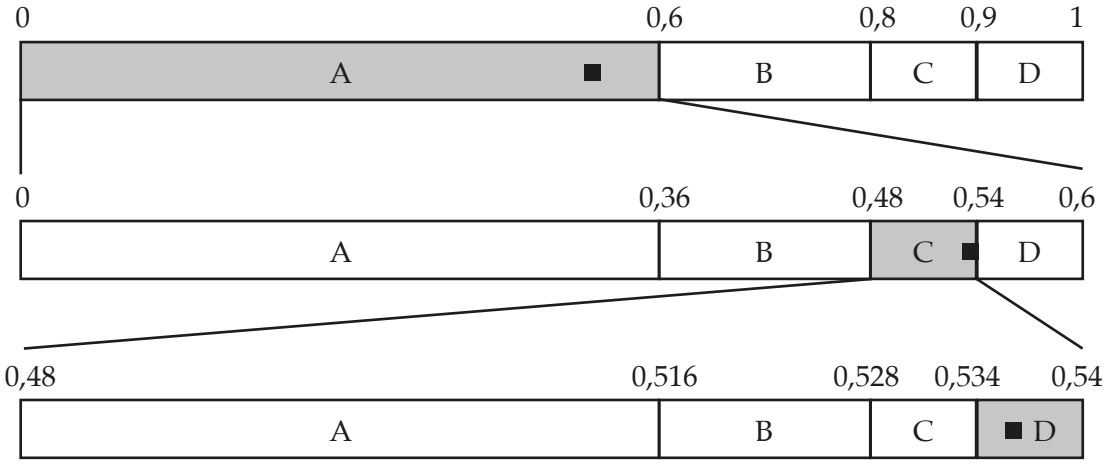
2.2 Aritmetické kódování

Druhý hlavní typ entropického kódování představuje aritmetické kódování, které vynalezl finský matematik Jorma Rissanen skoro o 30 let později než Huffmanovo kódování. [5] [6] Tento způsob kódování produkuje téměř optimální délky kódů. Hlavním rozdílem oproti Huffmanovu kódování je, že netvoří kódová slova pro každý vstupní symbol, ale místo toho zakóduje celou informaci do jednoho čísla z intervalu $\langle 0, 1)$. Princip kódování spočívá v přiřazení podintervalů pro každý vstupní symbol a jejich následné umístění do zmíněného intervalu $\langle 0, 1)$. Velikosti intervalů pro každý symbol jsou závislé na pravděpodobnosti výskytu daného symbolu, pro častější symbol bude vytvořen větší interval, a naopak pro méně častý symbol interval menší. Po určení těchto intervalů se pro symbol, který chceme zakódovat, vybere daný podinterval a podle něj musíme přerozdělit hlavní interval tak, že jeho hranice budou stejné jako hranice vybraného podintervalu. Nový interval ovšem musí zachovat hranice podintervalů pro všechny symboly ve stejném poměru jako na začátku. Tohle se provádí, dokud se nezpracují všechny symboly. Výsledkem kódování je číslo z posledního vybraného podintervalu (obvykle spodní hranice) a toto číslo se zapíše na výstup. Ukázka kódování je na obrázku 3.



Obrázek 3: Ukázka aritmetického kódování

Postup dekódování je velmi podobný kódování, s rozdílem, že nyní máme číslo v intervalu ⟨0, 1) místo posloupnosti symbolů. Kódovací tabulku s rozdělením intervalů pro všechny symboly musíme mít dostupnou z procesu kódování. Nyní najdeme podinterval, do kterého spadá vstupní číslo, podle toho zjistíme symbol, který zapíšeme na výstup. Poté se přerozdělí intervaly stejně jako v procesu kódování a opakujeme výběr podintervalu. Pro správné ukončení dekódovaní je potřeba nějaká dodatečná informace pro dekodér, například očekávaná délka informace. Jinak se může stát, že by dekodér přečetl více symbolů, než jsme ve skutečnosti zakódovali a došlo by k chybnému výsledku.



Obrázek 4: Ukázka aritmetického dekódování hodnoty 0,538

V reálném prostředí tohle opět nestačí, z důvodu omezené přesnosti desetinných čísel a kvůli zaokrouhlování vznikají nepřesné kódy nebo chyby. Obvykle se nepřesnost dá zmírnit zvětšením intervalu z ⟨0, 1) například na $\langle 0, 2^{32})$ (rozsah 32 bitů). Tímto získáme dostatečně velký interval, který zmenší míru nepřesnosti na přijatelnou hodnotu. Aritmetické kódování funguje velmi dobře a přibližně od roku 2000 se objevuje převážně v textových kompresorech, které dosahují velmi dobré míry komprese. Příkladem je program *PAQ* založen na algoritmu *Context mixing (CM)*, který pracuje s prediktorem následujících symbolů a aritmetickým kódováním. [7] Hlavní nevýhodou nejen tohoto konkrétního typu, ale i obecně aritmetického kódování, je relativně pomalá rychlost komprese a dekomprese díky procesorově náročným instrukcím jako násobení a dělení. Díky tomu bylo upřednostňováno Huffmanovo kódování všude, kde byla potřeba rychlého a efektivního přístupu, ačkoliv s ne úplně optimální mírou komprese.

3 Asymmetric numeral systems (ANS)

ANS je označení pro nové způsoby entropického kódování. Základ pro tyto metody představil poprvé v roce 2009 polský matematik Jarosław Duda ve své práci *Asymmetric numeral systems*. [3] Na základě této první verze vznikla kompresní metoda *ABS*, která je limitována pouze na abecedu s dvěma symboly (0 a 1). Proto je *ABS* konkurentem binárního aritmetického kódování, které je rychlé a efektivní. Kvůli tomu nebyl o ANS projeven velký zájem, jelikož se zdálo být nekompetitivní. O několik let později se o ANS znovu objevil zájem a spolu s upravenou verzí původní práce od Jarosława Dudy v roce 2014 vznikly dva nové způsoby kódování zaměřené především na abecedy s více než dvěma symboly. Tyto metody byly postupně rozlišeny jako *rANS* (range) a *tANS* (table).

Základní myšlenkou ANS je zakódování informace do jednoho přirozeného čísla x, které obsahuje $log_2(x)$ bitů informace. [8] Pracujeme-li obecně s jakoukoliv množinou symbolů s z množiny S, pro každý symbol s existuje jeho pravděpodobnost výskytu p_s. Pokud chceme přidat informace ze symbolu s k číslu x, dostaneme číslo x', pro které platí:

$$x' \approx x / p_s$$

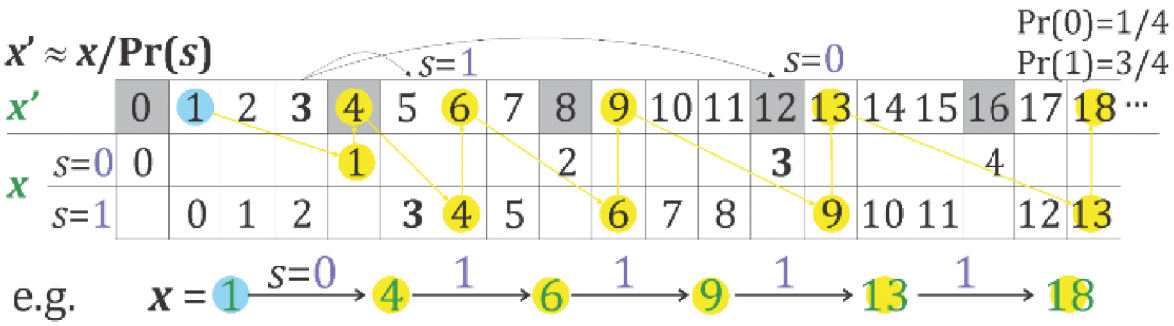
$$log_2(x') \approx log_2(x) + log_2(1/p_s)$$

kde $log_2(1/p_s)$ je počet bitů informace v symbolu s.

ANS se snaží najít kódovací a dekódovací funkce C a D, pro které platí:

$$C(s,x) = x', D(x') = (s,x)$$

V praxi lze tyto funkce definovat více způsoby, a to jak se k nim přistupuje mění typ kódování. Pro již zmíněný rANS nebo například rABS, což je intervalový kodér, který pracuje pouze s abecedou o dvou znacích (0 a 1), tyto funkce bývají definovány čistě jako matematické operace. V případě tANS se logika těchto funkcí vkládá do tabulky a vytváří se konečný automat. Rozdíly jsou hlavně v přístupu podobně jako u aritmetického kódování proti Huffmanovu kódování. Varianta rANS je podobná spíše aritmetickému/intervalovému (range) kódování, ale bývá rychlejší z důvodu méně násobení a dělení potřebných v hlavních krocích algoritmu. Varianta s konečným automatem tANS je spíše podobná Huffmanovu kódování a neobsahuje v hlavních krocích žádné násobení a dělení.



Obrázek 5: Základní princip ANS

Na výše uvedeném obrázku 5 lze vidět ukázku základního principu ANS pro binární abecedu. Zajímavým rANS kodérem je implementace *ryg-rans*, kterou vytvořil Fabian Giesen. [9] Nicméně tématem, kterým se diplomová práce zabývá je varianta tANS, konkrétně její první reálná implementace *Finite State Entropy* (*FSE*), kterou vytvořil Yann Collet. [10] Dnes se ANS kodéry prezentují jako velmi rychlé, dosahující rychlosti Huffmanova kódování a také účinné. Přibližují se optimálnímu počtu bitů na kódovaný symbol.

3.1 tANS

Principem pro tANS je, jak již bylo zmíněno, uložit veškerou logiku do tabulky a vytvořit konečný automat. Výhodou tohoto přístupu jsou relativně jednoduché operace, které tolik nezatěžují výpočetní kapacitu procesoru. Hlavní kroky jak kodéru tak dekodéru obsahují pouze bitové posuny a sčítání, díky tomu se rychlost tANS může velmi přiblížit rychlosti Huffmanova kódování. Algoritmus tANS má dvě zajímavé vlastnosti:

- Kódování a dekódování se provádí v opačném směru
- Kromě proudu bitů (zakódovaná data) existuje ještě hodnota *stav*

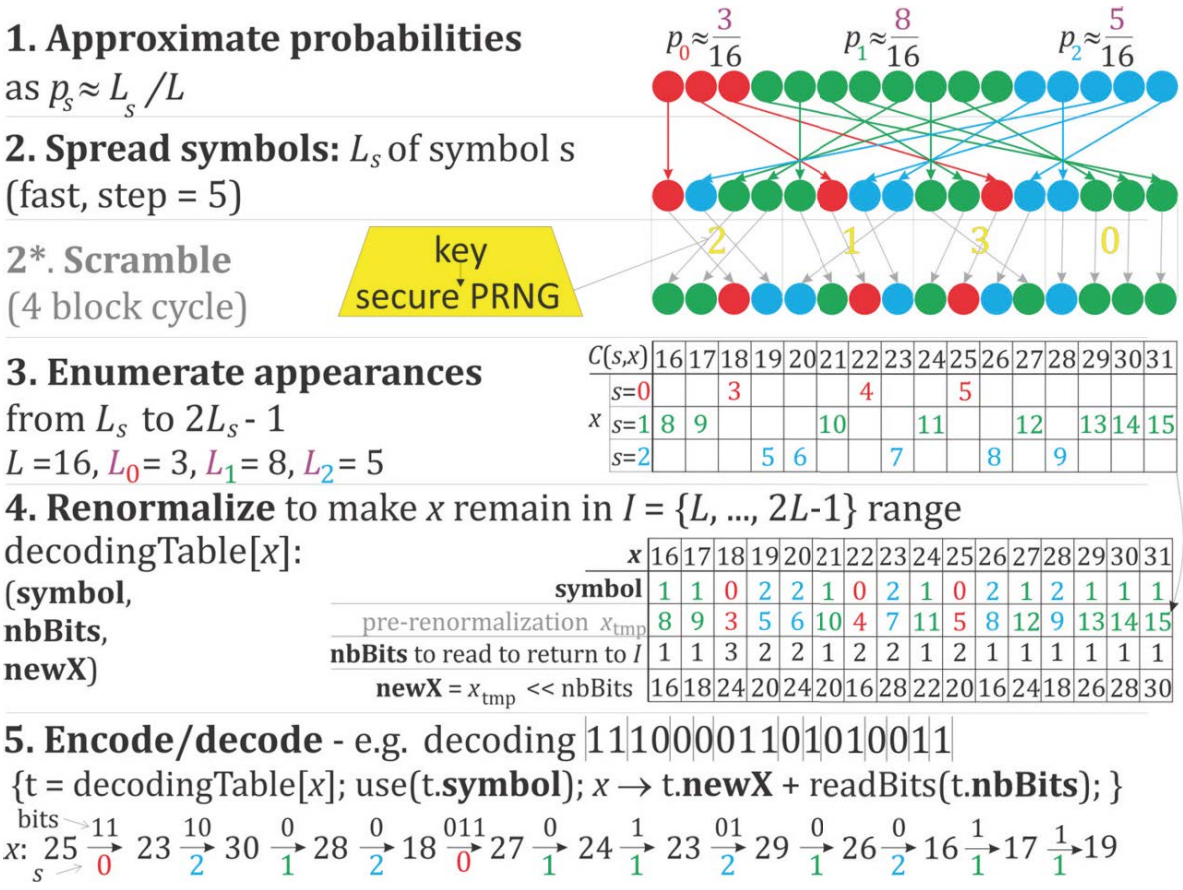
První vlastnost vychází z teorie ANS a prakticky provádění kódování a dekódování v opačném směru znamená, že se čtou zpracovávané soubory jednou od začátku a jednou od konce. Samotné operace v hlavních cyklech kódování a dekódování by se daly chápat jako opačné operace, na ty se ale podíváme později.

Hodnota *stav* udává aktuální stav v sestaveném konečném automatu. Tato hodnota může nabývat hodnot *0* až *L-1*, kde *L* je maximální počet stavů. Počet stavů může být proměnný, a později je jeden z testů zaměřen na zjištění ideální velikosti *L*. Yann Collet ve své FSE implementaci používá výchozí hodnotu 4096. [11] S počtem stavů *L* je také důležité znát bitový rozsah této hodnoty, který je později označen jako *R*, a platí: $L = 2^R$, pro $L = 4096$ je tedy $R = 12$.

Obecný postup tANS algoritmu lze popsat takto:

1. Určení pravděpodobností pro všechny symboly
2. Rozložení symbolů napříč všemi stavy
3. Očíslování pozic jednotlivých výskytů symbolů
4. Vytvoření dekódovací tabulky
5. Kódování/dekódování

Zjednodušeně je celý tento proces znázorněn na obrázku 6 níže. [12]



Obrázek 6: Obecný postup tANS algoritmu

Na obrázku 6 si lze všimnout ještě jednoho volitelného kroku, a tím je přeskupení symbolů podle určitého klíče po blocích stejné velikosti. Tímto krokem lze kromě komprese také dosáhnout zašifrování dat, a kdokoliv bez použitého klíče nebude schopen data dekódovat. Použitím šifrování získáme větší zabezpečení dat na úkor velmi malé náročnosti na procesor. Nicméně ztratíme nějakou účinnost komprese, jelikož jak bude ukázáno později, rozložení symbolů napříč stavy automatu má vliv na to, jak bude samotná komprese efektivní.

3.1.1 Dekódování

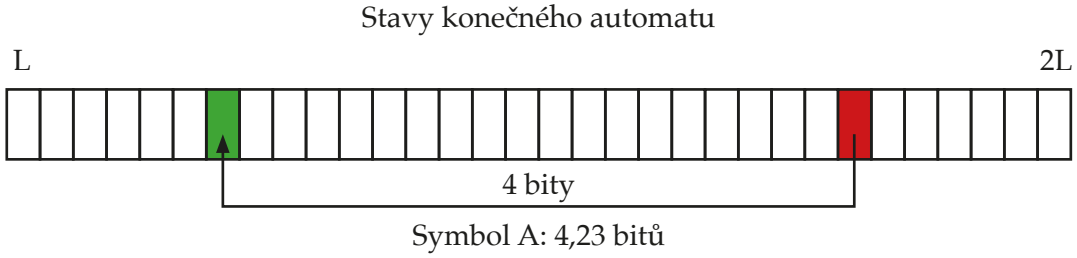
Dekódovací proces je oproti kódovacímu jednodušší, jelikož se zde přímo neuplatňuje teorie stojící za ANS. Díky tomu je hlavní krok dekódovaní poměrně obecný, a dal by se upravit pro použití například s Huffmanoým kódováním. Důležitou informací je, že hodnota stavu je dostačující k okamžitému dekódování jednoho symbolu. Jak již bylo zmíněno, stav je hodnota, která se udržuje mimo proud bitů, z něhož se pouze zjistí počáteční hodnota. Dalo by se říci, že pokud bychom byli schopni mít dostatečně velký počet stavů a k tomu dekódovací tabulku stejné velikosti, dekódování by bylo pouhé zapsání symbolu na výstup a přechod do dalšího stavu. Podle dekódovací tabulky totiž přesně známe symbol, který se má dekódovat a také následující stav. Ovšem tohle není reálně použitelné, především z dů-

vodu kapacity potřebné k vytvoření této tabulky, ale rovněž z důvodu příliš velké časové náročnosti. V dekódovací tabulce tedy nebude uložen rovnou následující stav, ale spíše základ, podle kterého nového stavu dosáhneme. Potřebujeme také znát počet bitů, které musí dekodér přečíst z proudu bitů pro daný stav a tato hodnota bude rovněž uložena v dekódovací tabulce. Dekódování poté vypadá následovně:

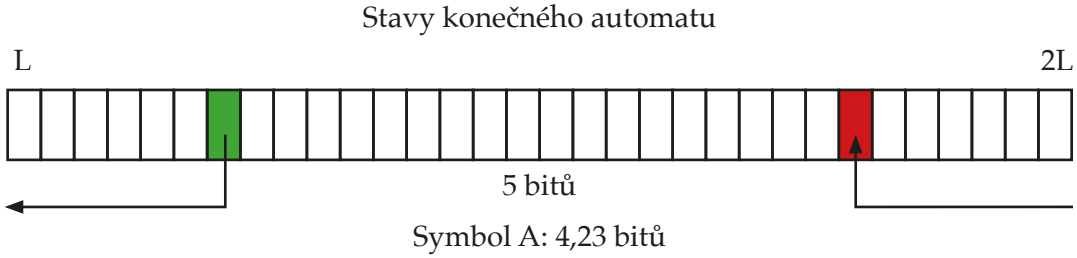
1. Dekódování symbolu pro aktuální stav
2. Přečtení určitého počtu bitů z proudu bitů
3. Přechod do nového stavu, který získáme součtem základu nového stavu a přečtených bitů

Počet přečtených bitů se dá chápat jako cena komprese, a závisí na četnosti výskytu daného symbolu. Například pokud je četnost výskytu pro symbol *A* rovna 4, a celkový rozsah je zmíněných 4096, znamená to, že k reprezentaci tohoto symbolu potřebujeme $-log_2(4/4096)$ bitů, neboli 10. Vycházíme z teorie, kterou popsal Claude E. Shannon a stejně tedy můžeme určit počet bitů pro jakkoliv četný symbol. Pokud se podíváme například na symbol o četnosti 5, potřebujeme k jeho reprezentaci $-log_2(5/4096) = 9{,}68$ bitů. Jedním z možných způsobů, jak dosáhnout požadovaného počtu bitů je rozdělení celého intervalu možných stavů na 5 (podle četnosti symbolu) podintervalů, jejichž velikost musí být mocninou čísla 2, o různých velikostech. Vzniknou tedy 2 intervaly o velikosti 9 bitů a 3 intervaly o velikosti 10 bitů (celkový součet zůstává 4096). Tyto podintervaly nyní pokrývají celý rozsah původních hodnot a každému výskytu se přiřadí začáteční pozice z jednoho intervalu (konkrétně 0, 512, 1024, 2048 a 3072), která tvoří dříve zmíněný základ potřebný pro přechod do dalšího stavu. Umístění a přiřazení těchto intervalů je také důležité pro výslednou účinnost. [13]

Povaha hodnoty stavu je taková, že se postupně zmenšuje a pokud překročí spodní limit, přečte se bit navíc a stav se opět dostane do vyšších hodnot z horního konce intervalu. Pokud pouze přecházíme z vysoké hodnoty stavu do nižší, bez překročení spodního limitu, není potřeba číst žádný bit navíc. Tento princip je znázorněn na následujících obrázcích 7 a 8.

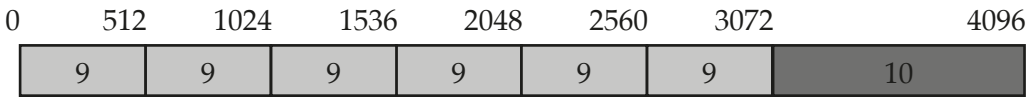


Obrázek 7: Snížení hodnoty stavu konečného automatu



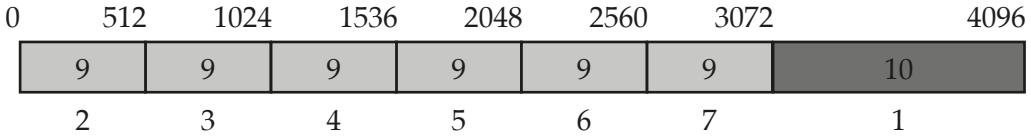
Obrázek 8: Překročení spodní hranice stavu konečného automatu

Podle principu práce se zlomky bitů lze předpokládat, že nižší hodnoty stavu jsou více pravděpodobné, a tedy podintervaly, které jsou umístěny na začátku celkového rozsahu mají větší váhu. Podintervaly jsou tedy uspořádány od menších (9 bitů) po větší (10 bitů). Pokud budeme chtít vytvořit podintervaly například pro symbol s pravděpodobností *7/4096*, budeme potřebovat šest podintervalů o velikosti 9 bitů a jeden o velikosti 10 bitů. Podintervaly budou seřazeny od nejmenších po největší. Příklad těchto podintervalů je na obrázku 9 níže.



Obrázek 9: Rozdělení rozsahu hodnot na podintervaly

Dříve již bylo zmíněno, že přečtení většího počtu bitů nastane v případě překročení spodní hranice rozsahu a jelikož se hodnota stavu postupně snižuje, největší šanci na toto překročení má symbol na nejnižší pozici. Přiřazení symbolů tedy musí začínat u větších podintervalů. V tomto konkrétním případě se začne u podintervalu o velikosti 10 bitů a k tomu připadá první pozice daného symbolu. Poté se pokračuje u podintervalů o velikosti 9 bitů. Čísluje se standardně od nejmenších hodnot po větší (v případě obrázku 9 se dá říci že zleva doprava). Výsledné přiřazení je znázorněno na obrázku 10 níže.



Obrázek 10: Přiřazení podintervalů pro jednotlivé výskyty symbolu

Toto rozložení je optimální vzhledem k potřebnému počtu bitů pro reprezentaci symbolu (9,19 bitů). Znamená to tedy, že pokaždé co tento symbol dekódujeme, tak spotřebujeme pouze zlomky bitů (konkrétně 0,19). Z toho vyplývá, že tento symbol můžeme dekódovat přibližně 5krát před tím, než bude potřeba přečíst o jeden bit více (10 místo 9). Tento postup je univerzální a platí obecně pro jakýkoliv symbol s libovolnou pravděpodobností. Jedná se o poslední část potřebnou k sestavení dekódovací tabulky, která ve výsledku obsahuje pro každý stav 3 informace:

- Dekódovaný symbol
- Počet bitů nutných pro přečtení z proudu bitů
- Základ pro hodnotu následujícího stavu

Důležitou částí nejen pro dekódovací proces je správné rozložení symbolů napříč všemi hodnotami stavu. První jednoduchou možností je seskupení symbolů a prosté naplnění tabulky. Tato metoda je v pozdější kapitole 5 testována vzhledem k výchozímu rozložení. Předpokládá se, že pokud se jednotlivé výskyty takto seskupí, dojde ke zbytečným ztrátám zlomkových bitů. Dříve bylo zmíněno, že stavy s nižší hodnotou jsou více pravděpodobné a znamenalo by to tedy nepoměr mezi symboly. Například symbol s pravděpodobností výskytu 33,3 % by měl být v tabulce umístěn na každé třetí pozici. Symbol s pravděpodobností

7,27 % na každé 13,76 pozici, což reálně představuje mezery mezi výskyty někdy 13 a jindy 14 pozic. Problém nastává, pokud více symbolů potřebujeme umístit na stejnou pozici, jelikož musíme jeden z těchto symbolů upřednostnit a ostatní posunout, čímž zase vznikají konflikty pro následující pozici. [14] [15]

Jeden z optimálních algoritmů je:

1. Pro každou pozici se porovnají všechny zlomkové hodnoty všech symbolů
2. Vybere se symbol s nejnižší hodnotou na danou pozici
3. K hodnotě tohoto symbolu se přičte $1/P$, kde P je pravděpodobnost výskytu symbolu
4. Opakuje se 1.–3. dokud není zaplněna celá tabulka

Tento přístup je téměř optimální, nicméně je časově relativně náročný, jelikož obsahuje velmi mnoho porovnávání. Algoritmus by se dal zrychlit například použitím přihrádkového řazení s větším počtem přihrádek než stavů. Přesně ideální rozložení symbolů ovšem není podmínkou pro správnou funkčnost, jelikož dochází pouze k malým ztrátám. Mnohem důležitější tedy je, aby symboly byly rozprostřeny přes celý rozsah možných hodnot. Metodu, která dosahuje dostačujících výsledků, definoval autor FSE Yann Collet takto [16]:

$$novýStav = (aktuálníStav + (5/8)\ L + 3)\ \%\ L$$

Pro implementaci této metody není potřeba žádného porovnání nebo řazení, pouze se pro všechny výskyty symbolů pomocí vztahu výše určí jejich pozice. Jedná se o velmi rychlou metodu, která sice nedosahuje úplně ideálního rozložení, ale její rychlost tento zápor převažuje.

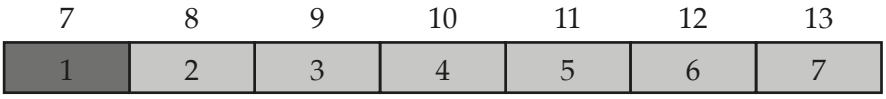
3.1.2 Kódování

V předchozí podkapitole 3.1.1 o dekódování byla zmíněna jedna velmi důležitá vlastnost ANS, a to že kódování a dekódování probíhá v opačných směrech. U entropických kodérů je toto neobvyklé a většinou se předpokládá, že kódování a dekódování probíhá stejným směrem. Také bylo zmíněno, že operace v kódování a dekódování by se daly chápat jako opačné. Začneme poslední operací v dekódování a tou je přechod do nového stavu pomocí součtu přečtených bitů a základem z dekódovací tabulky. Pokud dekodér čte určitý počet bitů z bitového proudu, tak opačnou operací je v kódování tyto bity zapsat. Bity, které se zapisují jsou vždy bity na spodních pozicích z hodnoty stav. Při dekódování počet těchto bitů známe z dekódovací tabulky, kterou při kódování zatím nemáme. Nicméně kodér má dostupnou jinou důležitou část informace, a tou je symbol, který má být zakódován. Stejně jako dekodér občas přečte n počet bitů a jindy $n+1$, tak kodér musí zapsat někdy n a jindy $n+1$ bitů. Pro přesné určení počtu bitů pro zápis se opět využije mapování podintervalů pro daný symbol. U dekódování bylo toto mapování znázorněno pro symbol s pravděpodobností výskytu $7/4096$. Podle těchto intervalů lze vidět, že pokud je hodnota stavu menší než 3072, počet bitů bude 9, a naopak pokud hodnota stavu je větší nebo rovna 3072, počet bitů bude 10. Při kódování je tedy nutné znát hodnotu této hranice pro každý symbol. Takto lze zjistit počet bitů pro zápis a tím pádem je jedna z částí kódování vyřešena. [17]

Zbývající informací je nový stav po kódování, respektive se jedná o stav předchozí, pokud se jedná o opačnou operaci z dekódování. Ke zjištění této hodnoty opět použijeme mapování podintervalů. Jelikož byly tyto podintervaly očíslovány (začínalo se u větších intervalů), můžeme přesně určit pozici symbolu, se kterou je daný podinterval spojen, a tato pozice je přesně ta hodnota, která chyběla pro kódování. Proces kódování jednoho symbolu se tedy dá popsat jako (známe aktuálně zpracovávaný symbol a stav) [18]:

1. Podle aktuálního stavu určíme pro kódovaný symbol počet bitů jako *n* nebo *n+1*
2. Tento počet bitů se zapíše do bitového proudu (spodní bity z aktuálního stavu)
3. Zjistí se konkrétní podinterval, ke kterému je přiřazen konkrétní výskyt symbolu
4. Automat přejde do nového stavu podle pozice symbolu

Zmíněné mapování podintervalů je velmi nepraktické, jelikož je potřeba pro každý symbol uložit jednotlivé intervaly s jejich velikostmi a také přiřazení pozic pro jednotlivé výskyty daného symbolu. Takové mapování by bylo poměrně náročné na paměť procesoru (L1 cache), a tím by také celé kódování a dekódování trvalo delší dobu. Implementačně je tedy vhodné toto mapování změnit tak, aby zabíralo co nejméně paměti. Všechny hodnoty z určitého podintervalu mají stejnou cílovou hodnotu nového stavu. Díky tomu můžeme tyto původní hodnoty zmenšit (konkrétně o 9 bitů) a získat tak mnohem menší celkový rozsah. Toto lze použít stejně pro všechny symboly s rozdílem o kolik budou tyto hodnoty zmenšeny. Ještě lepšího výsledku lze dosáhnout, pokud počáteční hodnotu každého podintervalu zmenšíme o takový počet bitů, jak velký je daný podinterval. Aby tato optimalizace fungovala, je potřeba mít hodnoty stavy ne z rozsahu ⟨0, L) (0 až 4095) ale posunuté do rozsahu ⟨L, 2L) (4096 až 8191). Jelikož se větší podintervaly zmenší o větší počet bitů, přesunou se na začátek nově vytvořeného mapování. Tímto dosáhneme rovněž očíslování podintervalů v pořadí, ve kterém jsou umístěny. Ukázka tohoto nového mapování pro symbol s pravděpodobností výskytu *7/4096* je na obrázku níže.



Obrázek 11: Efektivnější mapování podintervalů

Další vlastností, které si lze všimnou je, že první podinterval je označen stejně jako četnost výskytu daného symbolu, a další jsou inkrementovány. Struktura vytvořena tímto způsobem je mnohem přijatelnější jak pro implementaci, tak především pro zpracování procesorem. [19]

3.1.3 Porovnání s Huffmanovým kódováním

U dekódování bylo zmíněno, že je definováno spíše obecně a s určitými úpravami lze použít například také u Huffmanova kódování. Například pokud budeme mít symboly *A*, *B*, *C* a *D* s následujícími pravděpodobnostmi výskytu $P_A = 0{,}5$, $P_B = 0{,}25$, $P_C = 0{,}125$ a $P_D = 0{,}125$, lineární zápis rozhodovacího stromu pro tyto symboly by vypadal následovně:

0	1	2	3	4	5	6	7
A				B		C	D

Obrázek 12: Lineární zápis rozhodovacího stromu pro rozhodování od bitu na nejvyšší pozici

Takto vypadá klasická reprezentace, která při rozhodování postupuje od nejvyššího bitu. Pokud tedy na nejvyšším bitu je hodnota 0, víme že dekódujeme symbol *A*, posuneme porovnávaný blok o jeden bit vlevo a z proudu bitů načteme další bit na nejnižší pozici v bloku. Dále, pokud na nejvyšší pozici je bit s hodnotou 1, potřebujeme přečíst další bit, pokud je jeho hodnota 0, dekódujeme symbol *B* a tak dále. Nicméně můžeme k tomu přistoupit jiným způsobem, a to porovnáváním od nejnižšího bitu. Pokud tedy změníme pořadí bitů v rozhodovacím stromu, tak jeho lineární zápis bude vypadat takto:

0	1	2	3	4	5	6	7
A	B	A	C	A	B	A	D

Obrázek 13: Lineární zápis rozhodovacího stromu pro rozhodování od bitu na nejnižší pozici

Na první pohled je lineární zápis odlišný, ale rozhodovací stromy jsou ekvivalentní. Tato lineární reprezentace je zajímavá především z důvodu, že takové rozložení by vytvořila implementace tANS. Navíc má tato reprezentace následující vlastnosti:

- Základ hodnoty pro nový stav (v porovnání s tANS) nemusí být vypočítáván a uložen v dekódovací tabulce, ale lze jej získat ze samotné hodnoty aktuálního stavu
- Počet bitů, se kterými musí dekodér pracovat, je pevně daný pro každý symbol, pro symbol *A* to je 1 bit, pro symbol *B* jsou to 2 bity a pro symboly *C* a *D* jsou to 3 bity

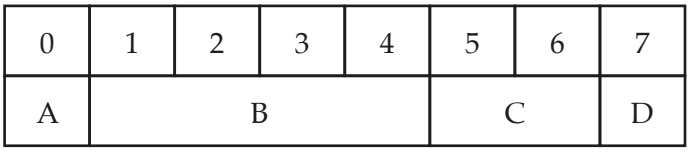
Dalo by se tedy říci, že tabulka lineární reprezentace Huffmanova kódování s rozhodováním od nejnižšího bitu je stejná jako tabulka, kterou by vytvořila implementace tANS, která má navíc pevně daný počet bitů na symbol a základ hodnoty pro následující stav. Ovšem je zde malý, ale důležitý rozdíl. Taková Huffmanova implementace by po dekódování zpracovávané bity posunula vpravo (bitovým posunem) a nové bity nahrála na nejvyšší pozice. Implementace tANS na rozdíl od toho neprovádí žádné bitové posuny, a tím pádem

neztratí žádné bity, ale k současné hodnotě stavu přičte nové bity (na nejnižší pozice). Tímto se vytváří závislosti na budoucích stavech, které jsou vyřešeny pomocí kódování a dekódování v opačném směru, jak již bylo dříve zmíněno. [20]

3.1.4 Porovnání s aritmetickým kódováním

Jak již bylo zmíněno v kapitole 2.2, hlavní myšlenka aritmetického kódování říká, že posunutí bitů je ekvivalentní dělení číslem 2. Proto se může tvrdit, že Huffmanovo kódování je zvláštním případem aritmetického – posunutí bitů je ekvivalentní dělení čísly mocniny 2. Na základě tohoto tvrzení je možné dělit jakýmkoli jiným číslem neomezeným pouze na mocninu čísla 2. Z toho vyplývá, že je možné dosáhnout zlomkových bitů. Může se dělit například číslem 1,11, což se dá přirovnat k ceně komprese 0,1 bitů.

Na následujícím obrázku je možné vidět posun (počáteční pozici) a rozsah každého symbolu, kde faktor je součet (počet hodnot) vydělený rozsahem. Aby bylo možné symbol zakódovat, musíme vždy vybírat z intervalu $\langle 0, 1)$, který se poté přerozdělí tak, aby odpovídal kódovací tabulce. V tomto případě je velikost tabulky 8 a hodnota z intervalu $\langle 0, 7)$.



Obrázek 14: Rozložení symbolů pro aritmetické kódování

Aby bylo možné zakódovat symbol *C*, je nutné určit nový rozsah. Z obrázku je zřejmé, že tento rozsah je nastaven na interval $\langle 5, 6)$. Nejprve se zakóduje 5 jakožto počáteční hodnota tohoto symbolu. Nyní zbývají pouze 2 pozice, aby bylo možné najít všechny možnosti, musí se tento rozsah rozšířit faktorem (vynásobit 4), tedy posunutím o 2 bity vlevo. Opět získáme rozsah 8 hodnot. V tuto chvíli je možné začít znovu, ale 2 bity jsou již určeny – jsou na vyšší bitové pozici (10). Tento příklad je velmi jednoduchý, protože zde figuruje pouze mocnina čísla 2, ale ten samý mechanismus lze použít na jakékoli rozložení symbolů a celý proces by fungoval úplně stejně. Je nutné počítat s chybami v zaokrouhlování, musí se nastavit ohraničená aritmetická přesnost, se kterou se musí počítat jak v kódování, tak dekódování (největším problémem jsou různé platformy).

Dekódování se provádí téměř stejně. V předchozím příkladě je nastavena dekódovací tabulka na známou velikost (8). Dekódování lze zapsat takto:

1. Načtení čísla
2. Dekódování symbolu (*C*, pokud je to 5 nebo 6)
3. Změna rozsahu
4. Přečtení dalších bitů

Celý proces funguje velmi dobře, ale je zde spousta operací násobení a dělení. Pokud se však zajistí, aby celkové rozložení symbolů byla mocnina 2, potom je možné snížit počet těchto operací na minimum.

U dekodéru tANS se začíná identicky – přečtený bit je ekvivalentní dělení 2. Prvním rozdílem je, že přečtení bitu není bitovým posunem. U Huffmanova nebo aritmetického dekódování je hodnota stavu čtena přímo z proudu bitů, zatímco u tANS je to hodnota udržovaná spolu s bitovým proudem. Pokud se narazí na symbol s pravděpodobností výskytu 50 %, tak tento symbol sníží hodnotu stavu o polovinu. Aby byla hodnota stavu v potřebném rozsahu, je nutné provést bitový posun a přečíst jeden bit. Dalším rozdílem je, že zlomkové bity se získávají načtením proměnného počtu bitů (pokud bude symbol zapsán 4,23 bity, dekodér někdy přečte 4, jindy 5 bitů). Nikdy tedy nemění rozsah o 10, jak to bylo u aritmetického dekodéru, ale někdy o 8, jindy o 16. [21]

3.2 Využití

Od roku 2014 se implementace různých variant ANS dostávají do reálně používaných programů. Mezi implementace patří například *Zstd* (Zstandard) [22], kterou vyvinul autor implementace FSE, tedy Yann Collet, pro Facebook. *Zstd* je spojením algoritmu typu *LZ* s entropickým kódováním, které poskytuje FSE [23]. Toto spojení se ukázalo jako velmi efektivní a začala ho používat také společnost Apple jako výchozí kompresní metodu v mobilních telefonech iPhone nebo počítačích Mac, nazvanou jako *LZFSE* (*Lempel-Ziv Finite State Entropy*) [24]. Implementace rANS také našla využití v zajímavém oboru, a tím je bioinformatika. Konkrétně se jedná o přidání rANS implementace do kompresní metody *CRAM* [25], využívanou například ke zpracování DNA informací.

Obecně lze tedy metody ANS považovat za velmi úspěšné, jelikož nejen že se prosadily v největších společnostech vůbec, ale také svoje výsledky potvrdily. V některých případech kompletně nahradily jak aritmetické, tak Huffmanovo kódování, především ve spojení s jinými metodami. Jedná se o velkou událost v oboru komprese dat, jelikož od aritmetického kódování nebyla nalezena metoda entropického kódování, která by přinesla téměř ideální míru komprese s velmi dobrými rychlostmi.

4 Implementace

Cílem mé práce byla implementace Finite State Entropy (FSE), což je již reálná implementace algoritmu Asymmetric Numeral Systems (ANS), konkrétně varianty s konečným automatem tANS (table). Z počátku byl díky jednoduchosti zvolen jako programovací jazyk C# pro vykoušení základních principů algoritmu, bez ohledu na optimalizaci. Vzniklé řešení bylo později převedeno do jazyka C++, pod kompilátorem MSVC++ ve vývojovém prostředí Visual Studio 2017. Jedná se tedy o vývoj na operačním systému Windows, nicméně v kódu nejsou žádné specifické funkce a případný převod pro jiný systém nebo kompilátor by neměl být problém. Výsledkem je konzolová aplikace, která přijímá dva vstupní parametry. První parametr je povinný a jedná se o název souboru, který chceme buď zakódovat, nebo dekódovat. Druhý parametr je volitelný a určuje počet stavů automatu neboli velikost kódovacích tabulek. Je udáván jako počet bitů, tedy pokud zadáme hodnotu 10, počet stavů bude 1024 (2^{10}). Program kontroluje příponu vstupního souboru, a podle toho určí, zda má jít o kódování či dekódování. Výstupem kódování je soubor s totožným názvem včetně přípony, ke kterému se přidá nová přípona *fsec*. Výstupem kódování je soubor s přidanou příponou *decoded*. Důvodem je zachování originálních souborů pro porovnání. Celý kód je dostupný online na portálu GitHub v repozitáři *hrtusd/fsec* (https://github.com/hrtusd/fsec).

4.1 Histogram symbolů

Vytvoření histogramu symbolů je první částí z přípravy dat ze vstupního souboru. V první části této funkce se binárně čte soubor po bajtech a v poli se zvětšuje počet výskytů daného symbolu. Indexem v poli je samotná hodnota symbolu. Implementace počítá s výchozím počtem symbolů 256 (hodnoty 0–255). V druhé části se funkce snaží zmenšit celkový počet symbolů z výchozí hodnoty 256. Kontrolují se tedy hodnoty výskytů symbolů od konce pole a pokud se tento symbol v souboru nevyskytuje (hodnota je rovna nule), tak se zmenší velikost o jeden symbol a kontroluje se předchozí. Může se tedy stát, že v některých souborech má maximální symbol hodnotu 190, takže 190–255 jsou nulové a tím pádem zbytečné pro další použití. Počet symbolů se tedy upraví na hodnotu 190 a dále se pracuje s touto hodnotou. Výsledkem této funkce je histogram výskytu symbolů v souboru, pokud možno tak snížená velikost abecedy z hodnoty 256 a také celková velikost souboru v bajtech.

```
unsigned long long countf(char* filename, int* &freqs)
{
      int size = symbol_count;
      unsigned long long true_sum = 0ULL;
      std::fstream ifs;

      ifs.open(filename, std::ios::in | std::ios::binary | std::ios::ate);
      true_sum = ifs.tellg();
      ifs.seekg(0, std::ios::beg);

      int blockSize = 1 << 12;
      std::vector<unsigned char> buffer(blockSize);
```

```
    int pos = 0;

    while (pos < true_sum)
    {
        if (pos + blockSize > true_sum) {
            blockSize = true_sum - pos;
            buffer.resize(blockSize);
        }

        ifs.read(reinterpret_cast<char *>(&buffer[0]), blockSize);

        for (int i = 0; i < blockSize; i++)
        {
            freqs[buffer[i]]++;
        }

        pos += blockSize;
    }

    ifs.close();

    // reduce total size from 256
    while (freqs[size - 1] == 0 && size > 0)
    {
        size--;
    }

    symbol_count = size;

    return true_sum;
}
```

Kód 1: Funkce pro vytvoření histogramu

4.2 Normalizace

Pokud chceme data ze souboru reprezentovat v dekódovací tabulce o velikosti L, musíme vstupní data nejprve normalizovat. Tímto procesem se upraví histogram výskytu symbolů tak, aby součet všech hodnot byl roven velikosti tabulky L. Obvykle tedy dochází ke zmenšení všech hodnot poměrem mezi velikostí tabulky L a velikostí vstupního souboru. V prvním kroku normalizace se tedy hodnoty pro všechny symboly vynásobí dříve zmíněným poměrem a zaokrouhlí na celá čísla. Musíme také zajistit, aby nikdy nedošlo k zaokrouhlení na nulu, jelikož bychom tímto nenávratně ztratili informaci. Po normalizaci je tedy vždy minimální hodnota 1, pokud se daný symbol vyskytoval v souboru. Toto řešení má ovšem na první pohled několik problémů. Prvním z nich je, že díky zaokrouhlování ztrácíme přesnost, která se ve výsledku projeví na míře komprese. Tato ztráta není obvykle tak velká, aby to činilo algoritmus neefektivním. Ideálním případem je, pokud k normalizaci nemusí vůbec dojít, což by znamenalo, že délka vstupního souboru je stejná jako velikost tabulky L, což je v reálných příkladech téměř nemožné. Na ukázce níže je příklad normalizace z celko-

vé velikosti $C = 10$ na velikost tabulky $L = 16$. V tomto případě se jedná o zvětšení hodnot, ale princip je totožný.

$$C_s = \{2; 3; 5\} \quad R = L/C = 16/10 = 1{,}6$$

$$F_s = \{2{\times}R; 3{\times}R; 5{\times}R\} = \{2{\times}1{,}6; 3{\times}1{,}6; 5{\times}1{,}6\} = \{3{,}2; 4{,}8; 8\} = \{3; 5; 8\}$$

Druhý problém normalizace je spojen se zaokrouhlováním a je řešen ve druhém kroku funkce. Může se totiž stát, že po upravení původních frekvencí symbolů nebude celkový součet roven velikosti L. Je nutné tedy upravit frekvence některých symbolů tak, aby se dorovnala tato odchylka. Musíme pamatovat na to, že při dalším upravování hodnot symbolů dochází opět k určité ztrátě přesnosti. Abychom tuto ztrátu co nejvíce minimalizovali, je nutné upravit hodnoty symbolů, které mají nejmenší chybovou odchylku. Odchylku tedy musíme spočítat pro všechny možnosti, a poté u symbolu s nejmenší odchylkou zvětšit či zmenšit hodnotu o 1. Výsledkem této funkce je pole o velikosti celkového počtu symbolů (výchozí hodnota 256, která se může zmenšit při vytváření histogramu) s normalizovanými hodnotami výskytů tak, aby se jejich součet rovnal velikosti L. [26]

```
int* normalize(int* &freqs, unsigned long long true_sum, int L)
{
	int* normalized = new int[symbol_count];
	int sum = 0;

	// up/down scale
	double scale = L / (double)true_sum;

	for (int i = 0; i < symbol_count; i++)
	{
		if (freqs[i] == 0) normalized[i] = 0;
		else
		{
			// scale every symbol count
			int f = (int)round(scale * freqs[i]);
			normalized[i] = (f > 0) ? f : 1;
			sum += normalized[i];
		}
	}

	// count sum difference
	int error = L - sum;

	while (error != 0)
	{
		int sign = error > 0 ? 1 : -1;
		double min = ULONG_MAX;
		int correct_idx = 0;
		for (int i = 0; i < symbol_count; i++)
		{
			double correction = freqs[i] *
				log2l((double)normalized[i] / (normalized[i] + sign));
```

```
                if (correction < min)
                {
                    correct_idx = i;
                }
            }
            normalized[correct_idx] += sign;
            error -= sign;
        }
        return normalized;
}
```

Kód 2: Funkce normalizace četností symbolů

4.3 Rozložení symbolů

Po normalizaci přichází klíčový krok, jelikož má velký vliv na výsledek komprese. Jedná se o rozložení symbolů napříč tabulkou velikosti L. Na tomto rozložení později závisí vytváření kódovacích tabulek, a čím lépe symboly umístíme, tím optimálnější komprese dosáhneme. Optimální rozložení je takové, aby se daný symbol nacházel v tabulce rovnoměrně v závislosti na jeho pravděpodobnosti výskytu. Například pokud symbol A má pravděpodobnost výskytu 50 % tak chceme, aby se po rozložení v tabulce vyskytoval na každém druhém místě. Symbol s pravděpodobností výskytu 25 % na každém čtvrtém místě a podobně. V reálných případech nelze přesného rozložení dosáhnout, jelikož symboly mezi sebou soutěží o určité pozice. Optimálnímu řešení se dá ve většině případů pouze přiblížit různými způsoby. V této implementaci byla zvolena heuristiku, kterou popisoval Yann Collet na svém blogu o implementaci FSE. Ta je v porovnání s jinými metodami méně optimální, ale rychlejší. Teoreticky rozdělí symboly pouze jedním průchodem celého pole bez složitého porovnávání a dopočítávání přesných pozic. Míra nepřesnosti, která použitím této metody vzniká opět není tak velká, aby se jednalo o neefektivní přístup. Výsledkem funkce je pole o velikosti L, v němž jsou rozloženy symboly podle jejich normalizovaných hodnot výskytu. [16]

```
int* spread(int* symbols, int L)
{
    int* symbol = new int[L];

    int idx = 0;
    int step = (int)(5.0 / 8 * L + 3);

    for (int i = 0; i < symbol_count; i++)
    {
        for (int j = 0; j < symbols[i]; j++)
        {
            symbol[idx] = i;
            idx = (idx + step) % L;
        }
    }
```

```
    return symbol;
}
```

Kód 3: Funkce pro rozložení symbolů

4.4 Dekódovací tabulka

Dekódovací tabulka je založena na rozložení symbolů a pouze doplní další nutné informace. V tomto kroku se už dá mluvit o konečném automatu se stavy X z $\langle L, 2L)$. Sestavení dekódovací tabulky je v celku jednoduché. Většinu potřebných informací máme již z rozložení symbolů, které nyní potřebujeme. Jedna z částí v této funkci je očíslování výskytů symbolů podle jejich četnosti. Například pokud se symbol A vyskytuje v tabulce třikrát, tak jeho výskyty budou očíslovány 3, 4 a 5. Tato hodnota je základ pro přechod ze současného do dalšího stavu. Jelikož tyto hodnoty jsou z intervalu $\langle 0, L)$ a automat má stavy z intervalu $\langle L, 2L)$, potřebujeme další hodnotu, která nám udává kolik bitů bude muset dekodér přečíst, aby se hodnota současného stavu posunula do intervalu $\langle L, 2L)$. Tyto informace jsou uloženy ve struktuře *decoding_entry*, která vypadá takto [27]:

decoding_entry:

- *symbol* – pro každý stav je jednoznačně určen dekódovaný symbol
- *next_state* – očíslovaný výskyt daného symbolu, základ pro následující stav
- *nb_bits* – počet bitů, které posunou hodnotu stavu do intervalu $\langle L, 2L)$ a tento počet zároveň čte dekodér

Výsledkem této funkce je pole struktur *decoding_entry*. Tato tabulka je později používaná v samotném kroku dekódování a k přesnému určení daného symbolu stačí pouze hodnota stavu, která se průběžně mění v závislosti na bitech, které jsou přečteny ze zakódovaného proudu bitů. Tato závislost není náhodná, jelikož jednotlivé kroky v dekódovacím cyklu lze chápat jako opačné akce z kódovacího cyklu. Například pokud v daném stavu víme, že máme přečíst z proudu bitů tři bity, tak to znamená, že při kódování jsou v tomto stavu tři bity zapsány.

```
decoding_entry* build_decoding_table(int* normalized, int L, int R)
{
    decoding_table = new decoding_entry[L];

    symbol_spread = spread(normalized, L);

    int* next = new int[symbol_count];
    for (int i = 0; i < symbol_count; i++)
    {
        next[i] = normalized[i];
    }

    for (int X = 0; X < L; X++)
    {
        decoding_table[X].symbol = symbol_spread[X];
```

```
            int x = next[decoding_table[X].symbol]++;
            decoding_table[X].next_state = x;
            decoding_table[X].nb_bits = (int)(R - floor(log2(x)));
      }

      return decoding_table;
}
```

Kód 4: Funkce pro vytvoření dekódovací tabulky

4.5 Tabulka symbolů

Na rozdíl od dekódování u kódování nám nestačí pouze jeden zdroj informací (dekódovací tabulka), ale potřebujeme zjistit ještě další informace o symbolech. Jednou z nich je očíslování výskytů stejně jako v sestavení dekódovací tabulky. Ačkoliv se ve výsledku jedná o stejné hodnoty, je zde rozdíl ve zpracování. Důležitou informací je počet bitů, které se mají pro daný symbol zapsat do bitového proudu. Tato hodnota je určena jako minimální počet a kodér se poté rozhoduje podle stavu, ve kterém se nachází automat, zda zapíše n bitů nebo $n+1$. Právě zmíněná hranice, se kterou kodér později pracuje je další informací specifickou pro daný symbol. Poslední informací je offset, který je potřebný pro sestavení kódovací tabulky. Tento offset nám zajistí správné umístění stavů ve výsledné kódovací tabulce. Hodnota offsetu začíná na $0 - F_s$, kde F_s je počet výskytů symbolu a postupně je k ní přičítán počet výskytů všech předchozích symbolů. Tyto informace jsou uloženy ve struktuře *symbol_entry*.

symbol_entry:

- *nb* – minimální počet bitů, které budou zapsány
- *k* – hranice podle které se může počet bitů pro zápis zvednout o jedna
- *start* – offset pro daný symbol
- *next* – očíslování výskytů symbolu, začíná na hodnotě F_s a později je při tvorbě kódovací tabulky zvětšována o jedna.

Jako výsledek této funkce je opět pole struktur *symbol_entry*, o velikosti počtu symbolů. Tato tabulka je spolu s optimálním rozložením symbolů nutnou informací k sestavení kódovací tabulky a také pro samotný kódovací cyklus.

```
symbol_entry* build_symbol_table(int* freqs, int L)
{
      symbol_entry* symbol_table = new symbol_entry[symbol_count];

      for (int i = 0; i < symbol_count; i++)
      {
            int fl = (int)floor(log2l((double)L / freqs[i]));
            if (freqs[i] == 0) fl = 1;

            symbol_table[i].nb = fl;
            symbol_table[i].k = (freqs[i] + freqs[i]) << symbol_table[i].nb;
```

```
            symbol_table[i].start = -freqs[i];

            for (int j = 0; j < i; j++)
            symbol_table[i].start += freqs[j];

            symbol_table[i].next = freqs[i];
      }

      return symbol_table;
}
```

Kód 5: Funkce pro vytvoření tabulky s informacemi o symbolech

4.6 Kódovací tabulka

Jak již bylo zmíněno v předchozích částech, k sestavení kódovací tabulky je potřeba pole s rozložením symbolů a tabulka s dodatečnými informacemi o symbolech. Podobně jako u sestavování dekódovací tabulky se zde pracuje s očíslovanými výskyty symbolů. Tvorba této tabulky spočívá v procházení pole s rozložením symbolů, podle kterého se zjistí offset a následující pozice (*next* ze struktury *symbol_entry*) a pokud tyto dvě hodnoty sečteme, dostaneme hodnotu stavu z intervalu $\langle 0, L)$, ze kterého se dostaneme do aktuálního stavu, jehož hodnota je v intervalu $\langle L, 2L)$. Je to tedy obdoba sestavení dekódovací tabulky, ale v tomto případě víme více informací o zpracovávaném symbolu. Výstupem funkce je pole o velikosti L, které obsahuje hodnotu následujícího stavu pro aktuální stav.

```
int* build_encoding_table(int* normalized, int L)
{
      encoding_table = new int[L];

      symbol_spread = spread(normalized, L);

      symbol_table = build_symbol_table(normalized, L);

      for (int x = L; x < 2 * L; x++)
      {
            int s = symbol_spread[x - L];
            int start = symbol_table[s].start;
            int next = symbol_table[s].next++;
            encoding_table[start + next] = x;
      }

      return encoding_table;
}
```

Kód 6: Funkce pro vytvoření kódovací tabulky

4.7 Bitové čtení/zápis

Jelikož v ANS potřebujeme pracovat s bity, respektive je číst a zapisovat do souboru, bylo nutné tento mechanismus implementovat. Prací s bity se zabývá struktura *bitstream*, která obsahuje několik členů především pro buffer. Struktura by se dala rozdělit do dvou hlavních částí, a to do jedné, která se zabývá přípravou souboru pro zápis a tedy kódování, a do části druhé, která naopak řeší čtení nutné při dekódování.

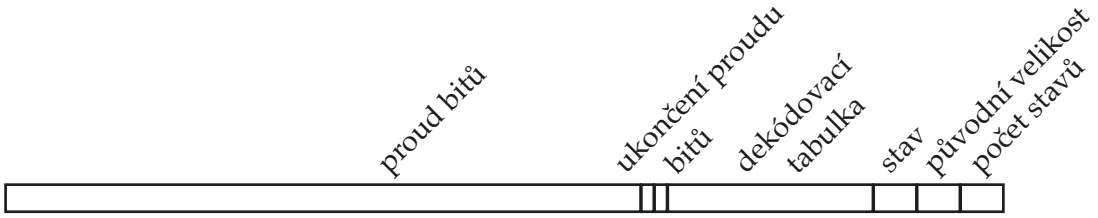
Při kódování jsou důležité především metody pro zápis zadaného počtu bitů, zápis dekódovací tabulky a ukončení *(flush)* souboru s posledními hodnotami. Zápis bitů je řešen pomocí vnitřního bufferu, který je plněn bity a v případě, že bylo zapsáno do bufferu více jak 8 bitů, tak je jeden bajt zapsán do souboru. Jelikož je pro práci se souborem použita třída *std::fstream*, tak není důležité zohlednit fakt, že zápis po jednotlivých bajtech je velmi neefektivní. Tato třída totiž vnitřně implementuje svůj buffer a neztrácíme tak důležitý výkon. Zapsání dekódovací tabulky je potom pouze zápis jednotlivých struktur z této tabulky do souboru. Než se zapíše dekódovací tabulka, musíme ukončit proud bitů bufferem, který obsahuje (nebo může obsahovat) zbývající bity a počet těchto bitů. Za dekódovací tabulku je poté ještě zapsán stav, ve kterém automat skončil (v tomto stavu bude začínat dekódování), celková délka originálního souboru a velikost automatu L.

```
void write2(unsigned int val, int nb)
{
      buffer <<= nb;
      buffer |= val;
      bitcount += nb;
      while (bitcount > 7)
      {
            unsigned char tmp;
            tmp = (unsigned char)(buffer >> (bitcount - 8));

            file.put(tmp);
            bitcount -= 8;
      }
}
```

Kód 7: Funkce pro bitové čtení

Jakmile začne dekódování, tak nejprve musíme přečíst tři hodnoty, které byly zapsány jako poslední při kódování, a poté samotnou dekódovací tabulku. Inicializují se buffery pro čtení a pozice v souboru se nastaví na konec proudu bitů vzniklého při kódování (začátek dekódovací tabulky). Struktura souboru je přiblížena na obrázku 15 níže.



Obrázek 15: Struktura zkomprimovaného souboru

Čtení bitů ze souboru je komplikovanější než jejich zápis. Jeden z důvodů je ten, že čtení musí probíhat opačným směrem než zápis, tedy od konce na začátek. To je vyřešeno manuálním posouváním pozice v souboru zpět před načtenou část. Druhým problémem je, že na rozdíl od zápisu, pokud chceme přečíst jeden bajt, tak dochází k velké ztrátě výkonu. Z tohoto důvodu je zde ještě druhý buffer, do kterého je načteno více bajtů najednou. Ty jsou potom zpracovávány do bufferu, ze kterého jsou postupně čteny bity (podobný způsob jako u zápisu).

```
unsigned int read2(int nb_bits)
{
     while (bitcount < nb_bits)
     {
          if (bufferRead == 0) {
               if (position < bufferSize) {
                    bufferSize = position;
                    position = 0;
               }
               else {
                    position -= bufferSize;
               }
               file.seekg(position, std::ios::beg);

               bufferVec.resize(bufferSize);
               file.read(reinterpret_cast<char *>(&bufferVec[0]), bufferSize);
               bufferRead = bufferSize;
          }

          int refreshRate = 2;
          int refreshed = 0;
          while (refreshed < refreshRate && bufferRead > 0)
          {
               int rotate = bitcount + refreshed++ * 8;
               unsigned char newByte = bufferVec[--bufferRead];

               bufferInternal |= newByte << rotate;
          }

          bitcount += 8 * refreshed;
     }

     unsigned int ret = bufferInternal & ((1 << nb_bits) - 1);
```

```
    bufferInternal >>= nb_bits;
    bitcount -= nb_bits;

    return ret;
}
```

Kód 8: Funkce pro bitové čtení

4.8 Kódování

Nyní se už dostáváme k samotnému procesu kódování. Abychom byli schopni úspěšně soubor zakódovat a později také dekódovat, musíme projít všemi částmi, které byly popsány dříve v této kapitole. Postup tedy vypadá takto:

1. Vytvoření histogramu
2. Normalizace frekvencí symbolů
3. Rozložení dat
4. Vytvoření kódovací tabulky spolu s tabulkou pro dodatečné informace o symbolech
5. Vytvoření dekódovací tabulky (nyní pro zápis do souboru)
6. Nastavení stavu na hodnotu L a inicializace proudu bitů
7. Přečtení symbolu z konce souboru
8. Zakódování symbolu, posunutí pozice v souboru směrem k začátku
9. Opakujeme 7. a 8. dokud jsme nezpracovali celý soubor
10. Ukončení proudu bitů, zapsání dekódovací tabulky a dalších hodnot

Jelikož máme k dispozici všechny funkce, tak jediné co zbývá, je samotný hlavní krok kódování. Nejprve zjistíme podle zpracovávaného symbolu počet bitů, které budeme zapisovat. Základ pro tuto hodnotu známe z tabulky symbolů, ze které rovněž víme hranici, která nám určuje, zda-li bude počet bitů n nebo $n+1$. Víme v jakém jsme aktuálně stavu, porovnáme ho tedy s touto hranicí a případně počet bitů zvětšíme. Následně tento počet bitů (spodní bity z hodnoty stavu) zapíšeme do proudu bitů. Zjistíme základ pro nový stav a podle kódovací tabulky přejdeme do stavu nového.

```
void encode(int symbol, int &state, bitstream &output)
{
    int nb_bits = symbol_table[symbol].nb;
    if (state >= symbol_table[symbol].k) nb_bits++;

    output.write2(state & ((1 << nb_bits) - 1), nb_bits);

    int start = symbol_table[symbol].start;
    int shift = (state >> nb_bits);
    int new_state = encoding_table[start + shift];
```

```
state = new_state;
}
```

Kód 9: Funkce kódování jednoho symbolu

4.9 Dekódování

Jelikož pro dekódování máme všechny informace obsaženy v zakódovaném souboru, nemusíme provádět takovou přípravu dat jako v případě kódování. Stačí nám pouze s pomocí struktury *bitstream* získat informace z dekódovaného souboru, a to především dekódovací tabulku, počáteční stav a celkovou délku souboru. Postup je tedy takový:

1. Inicializace struktur a proměnných potřebných k dekódování
2. Nastavení a připravení výstupního souboru
3. Dekódování jednoho symbolu
4. Zápis symbolu do výstupního souboru
5. Opakujeme 3. a 4. dokud jsme nedekódovali celý soubor
6. Ukončení výstupního souboru

Samotný hlavní krok dekódování jednoho symbolu je díky dekódovací tabulce velmi jednoduchý. Jelikož víme, ve kterém stavu se automat právě nachází, tak můžeme jednoznačně určit který symbol dekódujeme a zapíšeme na výstup (pro každý stav je v dekódovací tabulce přiřazen symbol). Stejně tak víme, kolik bitů musíme přečíst z proudu bitů. Z dekódovací tabulky také známe základ pro následující stav. Tuto hodnotu bitovým posunem vlevo posuneme tolikrát, kolik bitů jsme přečetli. Nakonec přičteme ony načtené bity a tímto získáme hodnotu nového stavu pro dekódování dalšího symbolu.

```
unsigned char decode(bitstream &input, int &state, decoding_entry* decoding_tab-
le, int L)
{
     decoding_entry de = decoding_table[state - L];

     int nb_bits = de.nb_bits;

     int shift = input.read2(nb_bits);

     int new_state = de.next_state;
     new_state <<= nb_bits;
     new_state |= shift;

     state = new_state;

     return (unsigned char)de.symbol;
}
```

Kód 10: Funkce dekódování jednoho symbolu

Z ukázky kódu 10 výše si lze všimnout, že se zde provádí operace, která by se dala přesunout do funkce vytváření dekódovací tabulky. Ušetřili bychom tak výpočetní kapacitu procesoru při samotném dekódování, které by tímto mohlo být o něco rychlejší. Jde o zmíněný bitový posun základu pro nový stav (*next_state*). K této operaci potřebujeme pouze počet bitů, které musíme přečíst, ale tato hodnota je také známa při vytváření dekódovací tabulky. Místo toho, aby v proměnné *next_state* byl uložen základ pro nový stav v intervalu $\langle 0, L)$, tak by tato hodnota byla přímo z intervalu $\langle L, 2L)$. Jedná se o jednu z drobných optimalizací, která by mohla především pro větší soubory snížit čas potřebný k dekódování.

4.10 Entropie, výpisy a časovač

Ve většině funkcí jsou také výpisy, které přesně ukazují, co se právě provedlo. Zobrazení těchto výpisů je podmíněno makrem *DEBUG_PRINT*, pokud je toto makro nastaveno na hodnotu 0, výpisy se nezobrazí, pokud je nastavena jiná hodnota, tak ano. Toto makro se nachází v hlavičkovém souboru *fsec.h*. Kromě těchto výpisů je v programu ještě jedna pomocná funkce, která slouží k výpisu všech sestavených tabulek (dekódovací, pro symboly, kódovací). Spolu tvoří velmi dobrý přehled o prováděných akcích, který velmi usnadnil vývoj a testování tohoto programu.

K dalším pomocným funkcím patří například funkce pro výpočet entropie, díky které můžeme odhadnout teoretický limit, pod který nelze soubor zakódovat. Tak jako existují u skoro každé akce informativní výpisy, tak je také měřena rychlost (respektive doba trvání) jednotlivých akcí. K měření času je použita struktura *high_resolution_clock* ze jmenného prostoru *std::chrono*. Kód potřebný k tomuto měření je v souborech *timer.h* a *timer.cpp*. Funkce pro výpočet entropie včetně měření času je na výpisu kódu 11 níže.

```
double entropy(int* &freqs, unsigned long long true_sum)
{
	printf_s("Calculating entropy - ");
	TimePoint start = timer_timepoint();

	double e = 0.0;
	for (int i = 0; i < symbol_count; i++)
	{
		double prob = (double)freqs[i] / true_sum;
		if (prob == 0) continue;
		double log = log2(prob);
		double mul = prob * log;
		e += mul;
	}

	TimePoint end = timer_timepoint();
	timer_print(start, end);

	return -e;
}
```

Kód 11: Funkce pro výpočet entropie dat

4.11 Ukázka

Pokud program spustíme v *Release* konfiguraci, informační výpisy jsou omezeny pouze na ty nejdůležitější. Mezi ně patří například časy jednotlivých kroků programu, informace o vstupním souboru, výsledný počet zapsaných bajtů, entropie nebo počet symbolů v abecedě. Jak takový výpis vypadá je zobrazeno na obrázku níže.

```
C:\Users\fxr\Documents\dp\fsec\fsec-v1>fsec.exe bench\enwik8
File: bench\enwik8

Encoding ...
Counting symbol frequencies - start
Counting symbol frequencies - 196 ms
Alphabet internal 241
File lenght (bytes): 101128023
Calculating entropy - start
Calculating entropy - 0 ms
Entropy: 5.111827
Normalizing symbol frequencies - start
Normalizing symbol frequencies - 0 ms
Building encoding table - start
Spreading symbol occurencies - start
Spreading symbol occurencies - 0 ms
Building symbol table - start
Building symbol table - 0 ms
Building encoding table - 0 ms
Building decoding table - start
Spreading symbol occurencies - start
Spreading symbol occurencies - 0 ms
Building decoding table - 0 ms
Bytes source: 101128023
Bytes written: 65702534
Bytes min: 64618618.005984
Ratio: 0.649697
Total time - 2672 ms

Done ...
```

Obrázek 16: Ukázka programu – kódování

Mezi další informace patří odhad minimálního počtu zapsaných bajtů, kterého můžeme dosáhnout. Tento odhad vychází z teorie informace, konkrétně souvisí s entropií vstupního souboru a je definován takto: *entropie* × *vstup* / *8*, kde *vstup* je celková velikost vstupního souboru v bajtech. Tato hodnota je na výpisu pojmenována jako *Bytes min*. Druhá zajímavá informace je poměr velikostí mezi výsledným a původním počtem bajtů. Je definována jednoduše jako podíl těchto velikostí, tedy: *výsledek* / *vstup*, kde *výsledek* je počet bajtů po kódování a *vstup* je stejně jako v předchozím případě velikost vstupního souboru v bajtech. Na výpisu jde o hodnotu *Ratio*. Po převedení na procenta (vynásobením 100) nám tato hodnota říká, kolik procent z původní velikosti je velikost po zakódování. V případě obrázku 16 výše je velikost souboru po zakódování přibližně 65 % z původní velikosti. Jinými slovy se původní soubor povedlo zmenšit o 35 % (100 − 65).

Dekódování je v tomto ohledu opět jednodušší, jelikož veškeré zpracování vstupních dat proběhne v kódování. Jedinou zajímavou informací je tedy pouze časová náročnost. Aby bylo dekódování zajímavější, museli bychom pustit program v konfiguraci *Debug* a také nastavit makro *DEBUG_PRINT* na hodnotu 1. Poté jsme schopni vidět jednotlivé kroky dekodéru a přechody mezi stavy. Pro reálné použití je to ovšem velmi nepraktické a pro přehlednost toto není doporučeno pro větší dekódovací tabulky (nad 16 stavů) a větší soubory než několik bajtů. Ukázka dekódování testovacího souboru o velikosti 10 bajtů a automatem s 16 stavy je na obrázku 17 níže.

```
Decoding ...
Reading state and decoding table - start
state 16
sum 10
start 106
bitc 7
Reading state and decoding table - 8 ms
Decoding - start
symbol: 65       state: 16 -> 26         bits: 2         nb: 3
symbol: 65       state: 26 -> 17         bits: 1         nb: 2
symbol: 66       state: 17 -> 23         bits: 3         nb: 2
read 62
symbol: 66       state: 23 -> 30         bits: 2         nb: 2
symbol: 66       state: 30 -> 19         bits: 1         nb: 1
symbol: 67       state: 19 -> 19         bits: 1         nb: 1
symbol: 67       state: 19 -> 19         bits: 1         nb: 1
symbol: 67       state: 19 -> 19         bits: 1         nb: 1
symbol: 67       state: 19 -> 18         bits: 0         nb: 1
symbol: 67       state: 18 -> 16         bits: 0         nb: 1
Decoding - 13 ms
File decoded to: plrabn12.txt.fsec.decoded
30 ms

Done ...
```

Obrázek 17: Ukázka programu – dekódování s podrobnými výpisy

Z výpisu vidíme jednak čtení z proudu bitů (řádek označený *start* je první hodnota v bufferu, řádky označeny *read* jsou následující čtení, pokud počet dostupných bitů v bufferu klesl pod potřebnou hodnotu), tak samotné přechody automatu mezi stavy. Každý řádek představuje dekódování jednoho symbolu, a tedy jeden přechod.

5 Testování

Testování programu bylo prováděno především za účelem zjištění efektivity a rychlosti. Na efektivitu, tedy jak moc je program schopen zkomprimovat soubory, má vliv hlavně optimální rozložení symbolů napříč dekódovací tabulkou a poté velikost (počet stavů) konečného automatu. Rychlost je nejvíce ovlivněna operacemi v hlavních kódovacích a dekódovacích cyklech, a také implementací práce se soubory. Jelikož se jedná o statistickou metodu komprese, která se v reálných implementacích využívá ve spojení se slovníkovou metodou, nemá porovnání s typickými kompresními programy jako ZIP nebo RAR příliš vypovídající hodnotu. Přínosnější je porovnání ve stejné kategorii, a to je například Huffmanovo, aritmetické nebo intervalové kódování.

5.1 Testovací sestava

Ke všem testům byla použita počítačová sestava s následující konfigurací:

- **Operační systém** – Windows 10 Pro verze 10.0.17763
- **Procesor** – Intel i5-6600K na frekvenci 4500 MHz
- **Základní deska** – ASUS Z170 PRO GAMING
- **Operační paměť** – G.Skill Trident Z 32 GB na frekvenci 3200 MHZ
- **Pevný disk** – SSD Samsung 970 EVO NVMe 250 GB

Grafická karta není v tomto případě relevantní.

5.2 Testovací soubory

Pro testování byly vybrány jak reálné, tak uměle vytvořené soubory. Jako reálné soubory byly zvoleny soubory *E.coli*, *bible.txt*, *world192.txt*, které spadají do kolekce *The Canterbury Corpus* [28]. Dalším reálným souborem je *enwik8* [29], což je prvních 10^8 bajtů textu z anglické wikipedie (konkrétněji ze zálohy *enwiki-20060303-pages-articles.xml*). Díky jeho velikosti lze lépe vidět vliv proměnných v programu na rychlost a kompresní poměr.

Pro soubory s uměle vytvořenými daty byl použit program *probagen* [23], jehož autorem je Yann Collet. Tento program umí vygenerovat podle zadaného parametru data tak, aby se v nich vyskytoval nějaký symbol se zadanou pravděpodobností. Takto bylo vygenerováno několik souborů a číslo v jejich názvu představuje procento pravděpodobnosti.

Soubor	Velikost [B]
E.coli	4 638 690
bible.txt	4 047 392
world192.txt	2 473 400
enwik8	101 128 023
proba1.bin	10 485 759
proba10.bin	10 485 759
proba25.bin	10 485 759
proba50.bin	10 485 759
proba75.bin	10 485 759
proba90.bin	10 485 759

Tabulka 1: Přehled velikostí testovaných souborů

5.3 Základní nastavení

Před specifickým testováním byly všechny soubory zkomprimovány (a dekomprimovány) se základním nastavením, abychom získali přehled a nějaký výchozí stav k porovnání. Základním nastavením je myšleno především počet stavů automatu, a to je 4096. Velikost tabulky také ovlivňuje rychlost programu, nicméně větším faktorem je velikost bloků, po kterých se načítají data ze souborů (případně zapisují). Ta je nastavena také na 4096, jelikož tato hodnota je zároveň doporučena jako výchozí hodnota pro alokaci prostoru na pevném disku (platí pro operační systém Windows 10).

Název	Velikost [B]	Entropie	CS [B]	CR [%]	CT [ms]	DT [ms]
E.coli	4 638 690	2	1 208 741	26,06	62	41
bible.txt	4 047 392	4,34	2 247 444	55,53	102	42
world192.txt	2 473 400	4,99	1 596 905	64,56	66	28
enwik8	101 128 023	5,11	65 751 698	65,02	2 236	1 101
proba1.bin	10 485 759	7,68	10 119 067	96,50	369	108
proba10.bin	10 485 759	4,7	6 205 841	59,18	251	106
proba25.bin	10 485 759	3,25	4 305 566	41,06	194	104
proba50.bin	10 485 759	2	2 669 619	25,46	144	93
proba75.bin	10 485 759	1,08	1 466 427	13,98	128	90
proba90.bin	10 485 759	0,52	734 004	7,00	87	85

Tabulka 2: Výsledky komprese se základním nastavením (**CS** = zkomprimovaná velikost, **CR** = kompresní poměr, **CT** = doba komprese, **DT** = doba dekomprese)

V tabulce 2 můžeme vidět přehled výsledků se základním nastavením programu. Udávaná doba jak komprese tak dekomprese je průměrná hodnota po 100 opakováních. Zkomprimovaná velikost je včetně dekódovací tabulky. Ve výchozím nastavení má dekódovací tabulka velikost 49 152 bajtů. Poměr je velikost zkomprimovaného souboru vůči

původní velikosti. Na první pohled jsou zajímavé výsledky u souborů *proba*, na které se více zaměříme v následujících testech.

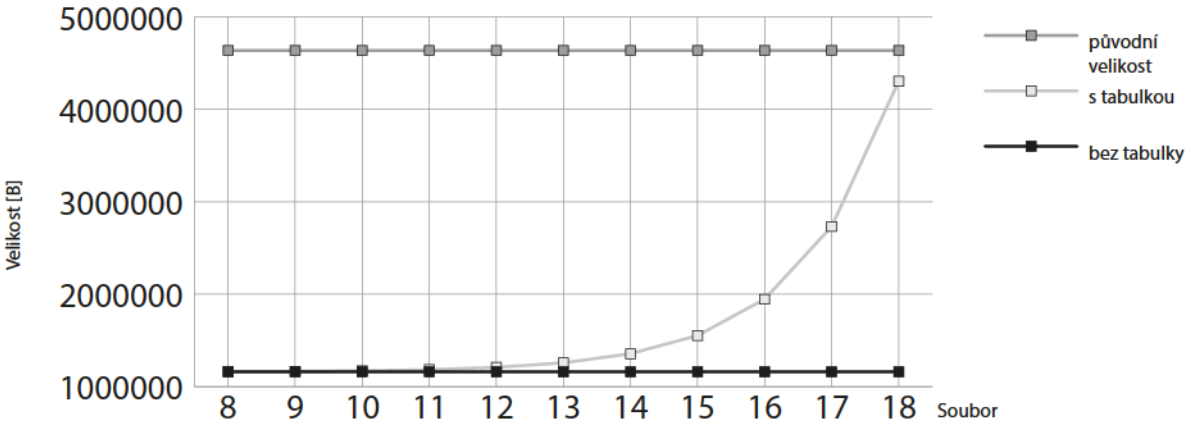
5.4 Počet stavů v automatu

Prvním ze specifických testů je zjištění optimální velikosti dekódovací tabulky. Teoreticky čím větší máme dekódovací tabulku, tím lepší komprese dosáhneme. Problém, který zde nastává, je nutnost práce s touto tabulkou. Jednak při běhu programu je tabulka uložena v operační paměti, a později je nutné tuto tabulku zapsat do souboru pro dekódování. V praxi musíme zvolit ideální poměr mezi velikostí zakódovaných dat a velikostí dekódovací tabulky. Pro tento test byly vybrány soubory *enwik8*, *E.coli* a *proba10.bin*. Tento test není zaměřen na rychlost komprese a dekomprese, a proto ji nebudu brát jako nevýhodu při větších velikostech dekódovací tabulky.

	E.coli		*enwik8*		*proba10.bin*	
velikost [B]	4 638 690		101 128 023		10 485 759	
počet stavů v bitech	**bez tabulky [B]**	**s tabulkou [B]**	**bez tabulky [B]**	**s tabulkou [B]**	**bez tabulky [B]**	**s tabulkou [B]**
8	1 160 210	**1 163 282**	89 488 609	89 491 681	6 512 059	6 515 131
9	1 159 583	1 165 727	78 653 938	78 660 082	6 269 703	6 275 847
10	1 159 589	1 171 877	71 333 680	71 345 968	6 192 240	6 204 528
11	**1 159 570**	1 184 146	67 395 401	67 419 977	6 164 726	**6 189 302**
12	1 159 577	1 208 729	65 702 534	65 751 686	6 156 677	6 205 829
13	**1 159 570**	1 257 874	64 989 631	65 087 935	6 156 458	6 254 762
14	1 159 571	1 356 179	64 683 820	**64 880 428**	6 156 292	6 352 900
15	1 159 571	1 552 787	64 638 672	65 031 888	6 156 263	6 549 479
16	**1 159 570**	1 946 002	64 629 267	65 415 699	6 156 262	6 942 694
17	1 159 571	2 732 435	64 623 291	66 196 155	**6 156 259**	7 729 123
18	1 159 571	4 305 299	**64 620 609**	67 766 337	6 156 261	9 301 989
limit [B]	1 159 569		64 618 619		6 156 069	
rozdíl [%]	0,01	67,74	24,59	24,34	3,39	29,68

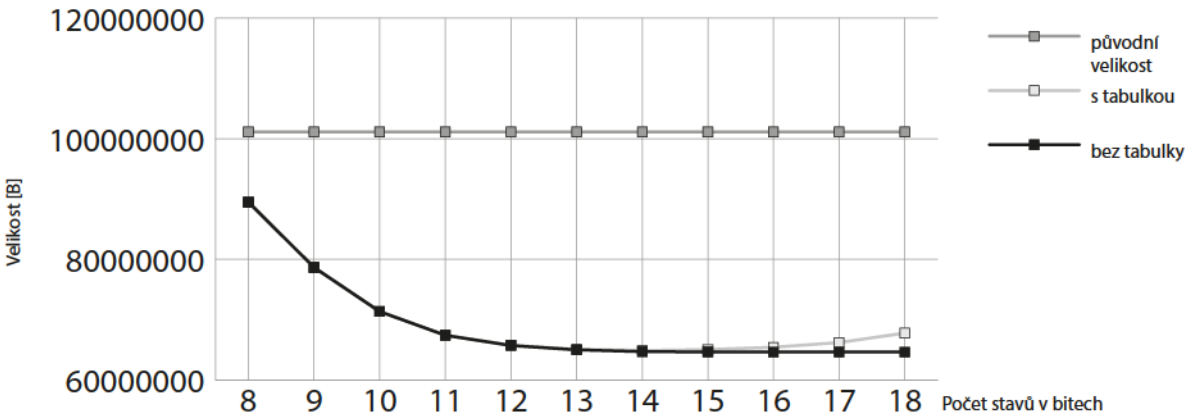
Tabulka 3: Vliv počtu stavů automatu na kompresi

V tabulce 3 jsou tučně označeny nejlepší výsledky pro daný soubor. Velikosti jsou dále rozděleny na dva sloupce, a to na velikost pouze zapsaných bitů a velikost včetně dekódovací tabulky. Z tabulky si můžeme všimnout, že se zvětšující se velikostí dekódovací tabulky opravdu klesá počet zapsaných bitů. U menších souborů jako *E.coli* nebo *proba10.bin* není tento pokles tak razantní, ale u většího souboru *enwik8* ho lze pozorovat lépe. Na obrázku 18 níže lze vidět průběh velikostí pro soubor *E.coli*. U tohoto souboru dochází k poklesu počtu zapsaných bitů pouze na začátku (menší počet stavů) a poté se prakticky nemění. Můžeme si také všimnout, že jsme velmi blízko odhadu limitu komprese a rozdíl je pouhé 2 bajty. Naopak celková velikost ihned od začátku roste a ke konci testu jsme téměř na stejné velikosti, kterou má původní soubor.



Obrázek 18: Průběh velikostí pro soubor *E.coli*

Soubor *enwik8* ukazuje odlišné chování. Ze začátku klesají obě velikosti a zlom, od kterého celková velikost začíná růst, je přibližně v polovině. Nicméně velikost pouze zapsaných bitů klesá až do konce a pokud bychom pokračovali s ještě většími počty stavů, pravděpodobně bychom se dostali ještě blíže k limitu, avšak opět na úkor velmi velkých dekódovacích tabulek. Tento průběh lze pozorovat na obrázku níže.



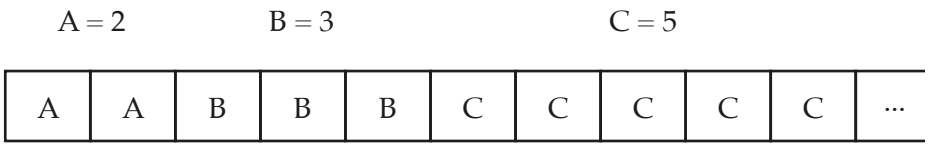
Obrázek 19: Průběh velikostí pro soubor *enwik8*

U souboru *proba10.bin* lze pozorovat podobný průběh jako u prvního souboru *E.coli*, avšak s větším poklesem na začátku a pozdějším růstem velikosti s dekódovací tabulkou. Optimální počet stavů pro tento soubor je 2^{11}. Pro soubor *E.coli* je to nejmenší možný obecný počet a to 2^8, a pro soubor *enwik8* bylo dosaženo nejlepších výsledků s počtem stavů 2^{14}.

Tímto testem byl potvrzen předpoklad, že se s rostoucím počtem stavů snižuje počet zapsaných bitů, a tedy zlepšuje komprese. Spolu s tímto bylo ukázáno, že není možné definovat optimální počet stavů napříč všemi soubory, jelikož záleží jednak na samotné velikosti souboru, ale také na datech, která tento soubor obsahuje. Jako výchozí počet stavů je v implementaci zvoleno 2^{12}, jelikož tímto vytváříme relativně malou dekódovací tabulku a získáme dostatečně dobrou míru komprese.

5.5 Rozložení symbolů

Dalším testem bylo zkoumání vlivu rozložení symbolů na účinnost komprese. V dřívějších kapitolách bylo zmíněno, že optimální rozložení symbolů napříč dekódovací tabulkou má pozitivní vliv na účinnost komprese. Implementace tohoto rozložení dále v textu, případně v tabulce bude označena jako standardní rozložení, jelikož je to výchozí implementace rozložení symbolů v programu. Tato metoda je porovnána s rozložením symbolů, kde všechny výskyty daného symbolu jsou seřazeny za sebou. Struktura je znázorněna na obrázku 20.



Obrázek 20: Znázornění naivního rozložení

Toto rozložení je později označováno jako naivní rozložení z důvodu, že se nejedná o žádnou sofistikovanou metodu, ale pouze prosté vyplnění. Pro tento test byly vybrány soubory *bible.txt*, *world192.txt*, *E.coli*, *enwik8*, *proba1.bin* a *proba90.bin*. Tímto je pokryto co největší množství různých typů souborů z celé testované kolekce. Pro každý soubor bylo testováno jak standardní, tak naivní rozložení symbolů, a to navíc pro automaty s počtem stavů 2^8 a 2^{12}. Jelikož nás zajímá vliv na účinnost komprese, výsledná velikost je uvedena bez velikosti dekódovací tabulky. Očekávaným výsledkem testu je v každém případě účinnější komprese.

soubor	počet stavů v bitech	standardní rozložení [B]	naivní rozložení [B]	rozdíl [B]	rozdíl [%]
bible.txt	8	2 366 350	2 405 745	39 395	1,64
	12	2 198 280	2 232 917	34 637	1,55
world192.txt	8	1 675 740	1 699 040	23 300	1,37
	12	1 547 741	1 564 962	17 221	1,10
E.coli	8	1 160 210	1 163 408	3 198	0,27
	12	1 159 577	1 163 400	3 823	0,33
enwik8	8	89 488 609	89 753 038	264 429	0,29
	12	65 702 534	66 574 001	871 467	1,31
proba1.bin	8	10 485 761	10 485 761	-	0,00
	12	10 069 903	10 141 651	71 748	0,71
proba90.bin	8	695 461	709 287	13 826	**1,95**
	12	684 840	697 134	12 294	1,76

Tabulka 4: Vliv rozložení symbolů na účinnost komprese

V tabulce 4 lze vidět přehled všech výsledných velikostí (bez dekódovací tabulky) pro zmíněné soubory, metody rozložení symbolů a velikosti automatů. Pro lepší orientaci slouží sloupce s rozdílem, které porovnávají účinnost jednotlivých způsobů. Tučně je označen nejlepší výsledek testu (největší rozdíl).

Výsledky testu potvrdily očekávání a kromě jediného případu, kde byl rozdíl nulový, byly velikosti zkomprimovaných dat menší s metodou standardního rozložení symbolů, která je výchozí implementací. Zlepšení se pohybovalo od 0,27 % po 1,95 % a ačkoliv to určitě není zanedbatelná hodnota, byly očekávány výsledky spíše mezi 2 % – 5 %. Jelikož je časová náročnost téměř totožná (oběma metodám stačí jeden průchod přes celou množinu symbolů), není důvod standardní rozložení symbolů nepoužít.

5.6 Porovnání s jinými algoritmy

Metoda FSE (neboli tANS) spadá do kategorie statistických kodérů, a tak jsou jeho přímými konkurenty Huffmanovo kódování nebo aritmetické kódování. V běžných kompresních programech se setkáváme s kombinacemi slovníkových a statistických metod, a není tedy příliš přínosné provádět porovnání například s programy jako ZIP nebo RAR. Proto byla vybrána k testování jedna implementace Huffmanova kódování [30] a zároveň pro soubory *bible.txt*, *world192.txt* a *E.coli* byly vybrány výsledky z testování implementace aritmetického a intervalového kódování [31]. Tento test srovná výsledné velikosti třech zmíněných souborů ve všech čtyřech metodách komprese. V druhé části testu poté bude bližší zaměření na rozdíly mezi Huffmanovým kódováním a vlastní implementací. K tomu jsou použity všechny soubory *proba.bin* vygenerované programem *probagen*.

Očekávaný výsledek první části testu je dosažení nejmenší zkomprimované velikosti pomocí aritmetického nebo intervalového kódování. V druhé části by měla lepších výsledků dosáhnout vlastní implementace FSE – *fsec*.

	E.coli	*world192.txt*	*bible.txt*
původní velikost [B]	4 638 690	2 473 400	4 047 392
aritmetické kódování [B]	1 160 066	**1 545 742**	**2 197 536**
intervalové kódování [B]	1 160 511	1 546 045	2 197 987
Huffmanovo kódování [B]	**1 159 688**	1 558 723	2 218 539
fsec [B]	1 163 294	1 576 181	2 225 171

Tabulka 5: Přehled výsledků statistických kodérů

V tabulce můžeme vidět výsledné velikosti celých souborů po kompresi danou metodou. Nejleší výsledky pro daný soubor jsou označeny tučně. Pro soubory *world192.txt* a *bible.txt* dosáhl podle očekávání nejlepších výsledků aritmetický kodér. Překvapivě pro soubor *E.coli* bylo nejúčinnější Huffmanovo kódování, pravděpodobně kvůli struktuře souboru. Vlastní implementaci *fsec* v tomto testu škodí neoptimální velikost dekódovací tabulky. Ta by se totiž po optimalizaci mohla ze své velikosti snížit alespoň o polovinu a poté by se mohly výsledky více přiblížit, či dokonce překonat ostatní implementace.

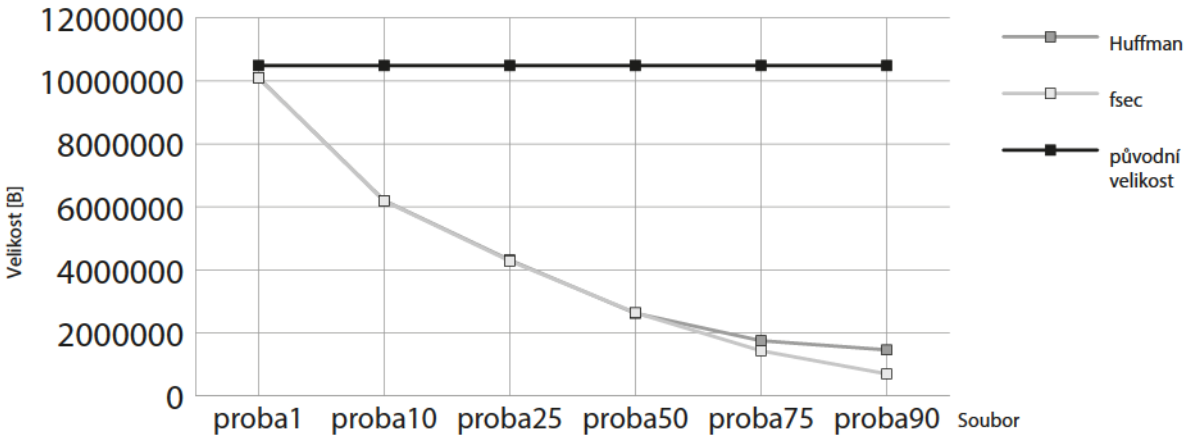
Kde by ale opravdu měla implementace *fsec* ukázat své silné stránky, jsou soubory, ve kterých se nějaký symbol vyskytuje s velkou pravděpodobností (větší než 50 %). Podíváme

se tedy na výsledky testu pro soubory *proba.bin*, které jsou zkomprimovány vlastní implementací *fsec* a implementací Huffmanova kódování.

soubor	velikost [B]	fsec [B]	Huffman [B]	rozdíl [%]
proba1.bin	10 485 759	10 099 204	10 106 433	0,07
proba10.bin	10 485 759	6 189 314	6 201 520	0,12
proba25.bin	10 485 759	4 282 899	4 309 704	0,26
proba50.bin	10 485 759	2 635 987	2 620 435	-0,15
proba75.bin	10 485 759	1 427 339	1 747 099	3,05
proba90.bin	10 485 759	694 862	1 456 650	7,26

Tabulka 6: Porovnání metody fsec a Huffmanova kódování na souborech s kontrolovanou pravděpodobností výskytu symbolů

V tabulce 6 lze vidět jednak jednotlivé velikosti souborů pro danou metodu, tak hlavně rozdíl, který porovnává zlepšení (či zhoršení) vlastní implementace *fsec* vůči implementace Huffmanova kódování. Huffmanovo kódování dosáhlo lepších výsledků v případě souboru *proba50.bin*. Pro ostatní soubory byla účinnější vlastní implementace *fsec*, především v souborech s vysokou pravděpodobností výskytu daného symbolu. Čím větší je pravděpodobnost výskytu symbolu v souboru, tím je rozdíl těchto metod větší. Důvodem je omezení Huffmanových kódů na minimální délku jednoho bitu. FSE toto omezení nemá a díky tomu je kódování velmi častých jevů silnou stránkou tohoto algoritmu. Průběh velikostí pro obě metody lze vidět na obrázku 21 níže.



Obrázek 21: Porovnání efektivity Huffmanova kódování a vlastní implementace FSE

Očekávání testu se z větší míry naplnila. Jediným nečekaným výsledkem bylo nejúčinnější kódování pro soubor *E.coli*, kterým se stalo Huffmanovo kódování. V druhé části testu se potvrdila silná stránka algoritmu FSE a tou je kódování velmi častých jevů s velmi dobrou přesností bez zmenšení rychlosti.

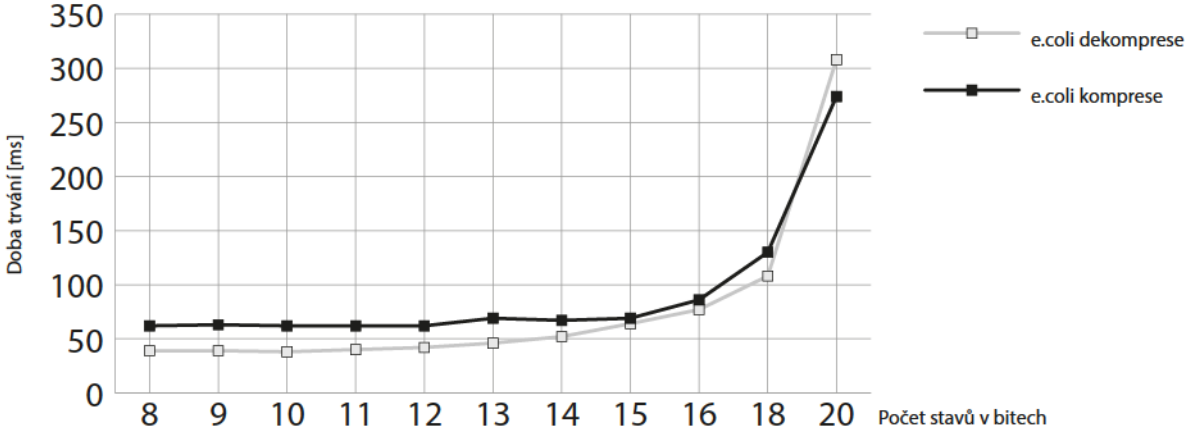
5.7 Rychlost

Posledním specifickým testem je testování rychlosti komprese a dekomprese vlastní implementace. Na rychlost mají (kromě optimalizací) vliv zejména velikost konečného automatu a velikost bloku pro zápis a čtení souboru. V tomto testu byly oba tyto faktory porovnány nezávisle na sobě. Pro testování rychlosti v závislosti na počtu stavů v automatu byly zvoleny soubory *E.coli* (téměř nejmenší z kolekce) a *enwik8* (největší z kolekce). Díky řádově odlišným velikostem těchto souborů se očekávají lehce jiné výsledky, nicméně by se měla změna rychlosti projevit stejně u obou souborů. V druhé části testu poté byla měřena změna rychlosti v závislosti na velikosti bloku pro práci se soubory. K tomuto testu byl použit pouze největší soubor *enwik8*, na kterém by měla být změna v rychlosti vidět nejvíce.

	enwik8				*E.coli*			
počet stavů v bitech	**komprese [ms]**	**nárůst [%]**	**dekomprese [ms]**	**nárůst [%]**	**komprese [ms]**	**nárůst [%]**	**dekomprese [ms]**	**nárůst [%]**
8	3 357	0	**978**	0	**62**	0	39	0
9	3 020	-10	1 059	2	63	2	39	0
10	2 793	-17	1 051	0	**62**	0	**38**	-3
11	2 712	-19	1 058	0	**62**	0	40	3
12	2 619	-22	1 121	0	**62**	0	42	8
13	2 616	-22	1 171	11	69	11	46	18
14	**2 615**	-22	1 301	8	67	8	52	33
15	2 645	-21	1 590	11	69	11	64	64
16	2 763	-18	1 847	39	86	39	77	97
18	3 059	-9	2 284	110	130	110	108	177
20	3 595	7	5 266	342	274	342	308	690

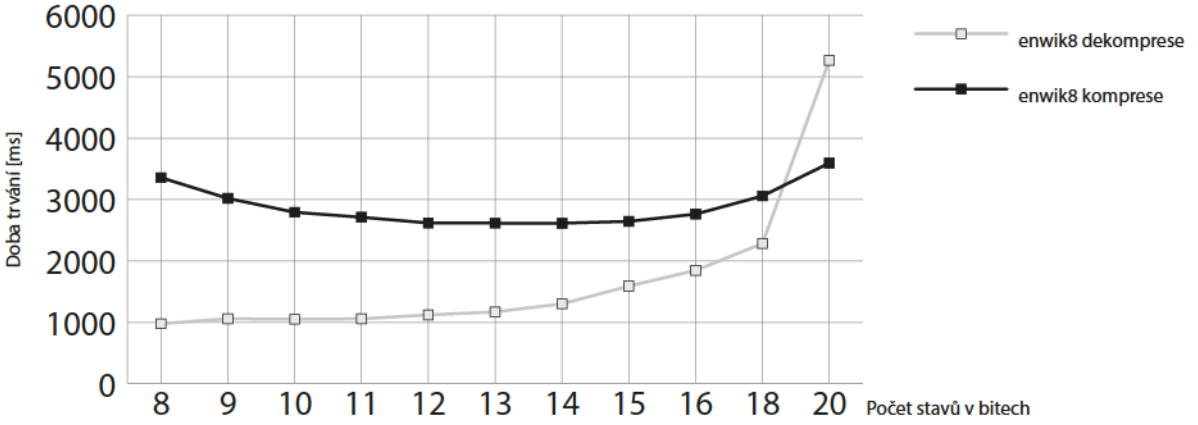
Tabulka 7: Rychlost komprese a dekomprese v závislosti na počtu stavů automatu

V tabulce 7 můžeme vidět výsledné rychlosti jak pro kompresi, tak pro dekompresi, testovaných souborů *enwik8* a *E.coli*. Rychlosti jsou uváděny v milisekundách a představují průměrnou hodnotu po 100 měřeních. Tučně jsou zvýrazněny nejlepší dosažené výsledky pro všechny možnosti. Ve sloupci nárůst lze pozorovat procentuální zvýšení dané doby oproti prvnímu času (pro počet stavů 2^8). Výsledky souboru *E.coli* představují očekávaný průběh, a to z počátku velmi mírný růst a přibližně od počtu stavů 2^{14} prudké zvětšení doby potřebné pro kompresi i dekompresi. Nejlepší velikosti pro tento soubor jsou ze začátku testovací množiny. Průběh pro tento soubor lze vidět na obrázku 22 níže.



Obrázek 22: Závislost rychlosti na počtu stavů pro soubor *E.coli*

Zajímavostí je průběh tohoto testu pro soubor *enwik8*. Zde totiž od začátku docházelo k poklesu doby potřebné pro kompresi, kdy ke zlomu došlo okolo počtu stavů 2^{13} 2^{14} a od této doby se doba trvání zvětšovala, tak jak se předpovídalo. Průběh doby dekomprese je podle očekávání a podobá se průběhu pro soubor *E.coli*. Celkový průběh lze opět pozorovat na obrázku 23 níže.



Obrázek 23: Závislost rychlosti na počtu stavů pro soubor *enwik8*

Výsledky první části jsou z jedné strany podle očekávání, a to že s rostoucím počtem stavů v automatu také roste doba potřebná k dekódování. Pro kódování byl stejný předpoklad, ale u souboru *enwik8* lze vidět, že se tento předpoklad nepotvrdil. Důvod z testování není zřejmý, nicméně obecně nemůžeme s jistotou říct, že čím větší počet stavů, tím déle bude komprese trvat.

Druhá část testu se netýká konkrétně algoritmu FSE, ale jedná se o obecné testování rychlosti implementace v závislosti na velikosti bloku pro čtení nebo zápis do souboru. Z počátku implementace pracovala pouze s velikostí jednoho bajtu, což bylo pro vývoj a první testování dostačující, ale v pozdějších testech se to projevilo jako velmi nedostačující a omezující faktor. Po implementaci práce s větším blokem dat, byla jako výchozí hodnota nastavena na 2^{12} (4096). Tento test by měl ukázat jaký přínos má použití většího bloku, pro-

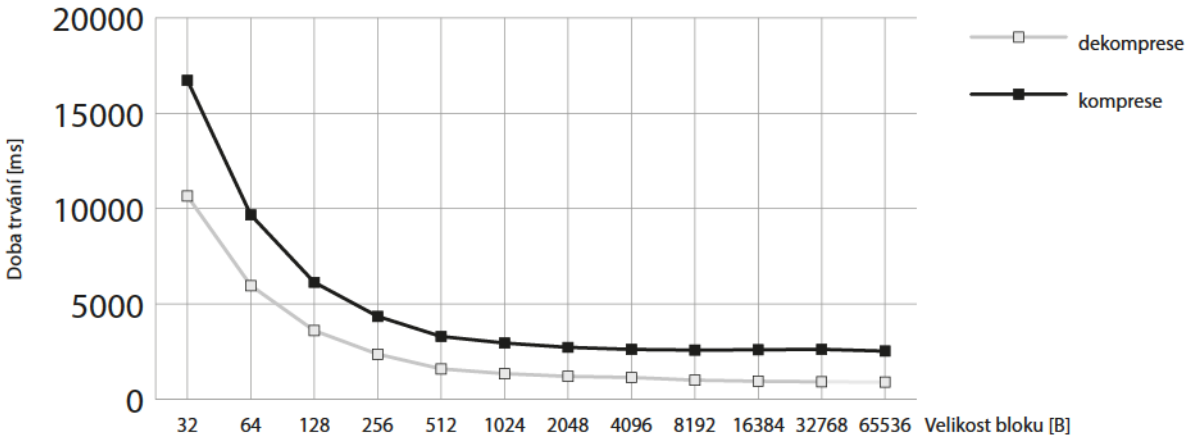
tože stejně jako pokud zvětšujeme počet stavů automatu, a tím pádem velikost dekódovací tabulky, roste také paměťová náročnost programu.

Pro test byly zvoleny velikosti bloku od 32 B po 65536 B, tedy od 2^5 do 2^{16}. Programové nastavení bylo na výchozích hodnotách, konkrétně především počet stavů byl 4096. Pro každou velikost bloku proběhlo 100 cyklů komprese a 100 cyklů dekomprese. Uváděné časy jsou jako průměrné hodnoty z těchto měření.

velikost bloku [B]	komprese [ms]	dekomprese [ms]
32	16 731	10 659
64	9 672	5 966
128	6 137	3 604
256	4 351	2 370
512	3 296	1 598
1 024	2 952	1 347
2 048	2 726	1 204
4 096	2 617	1 147
8 192	2 581	1 005
16 384	2 599	949
32 768	2 621	918
65 536	2 529	904

Tabulka 8: Doba komprese a dekomprese pro různé velikosti bloku

Podle tabulky je na první pohled zřejmé, že se předpoklad testu potvrdil a s rostoucí velikostí bloku dat se snižuje doba trvání jak komprese tak dekomprese. Můžeme si také všimnout, že přibližně od velikosti 2048 nedochází u doby komprese k tak velkým zlepšením. U dekomprese je vidět ještě relativně velké zrychlení až do konce testovaných hodnot. Průběh dob trvání lze vidět na obrázku 24 níže.



Obrázek 24: Závislosti dob komprese a dekomprese na velikosti bloku

Výsledek této části testu opět potvrdil předpoklad, kdy se potřebná doba pro kompresi a dekompresi sníží, pokud se zvětší velikost bloku pro zpracovávaná data zvětší. Tyto výsledky se mohou lišit pro různé operační systémy, respektive různé souborové systémy či pro různé modely pevných disků. Momentálně se na základě tohoto testu dá říci, že minimální doporučená velikost bloku je 2048 nebo 4096, avšak program dokáže dobře využít i větší bloky o velikosti 32 kB (32768 B) nebo 64 kB (65536 B).

5.8 Zhodnocení

Testování potvrdilo mnoho předpokladů o nastavení programu, ale také přineslo několik překvapivých výsledků. Zvolené základní nastavení programu lze považovat za optimální, jelikož především počet stavů v automatu nelze nastavit optimálně pro všechny soubory. Výchozí hodnota pro tento parametr je 4096, ale můžeme ji změnit pomocí parametru programu. Druhá hodnota, kterou ovšem nelze změnit při spuštění programu, ale je nastavena makrem přímo ve zdrojovém kódu, je velikost bloku pro práci s daty ze souboru (čtení a zápis). Tato hodnota je, stejně jako počet stavů v automatu, nastavena na 4096. Nicméně ukázalo se, že program může na testované sestavě fungovat lépe i s většími bloky jako například 32768 nebo 65536.

Hlavní slabinou implementace je neoptimalizovaná velikost dekódovací tabulky, která by v ideálním případě mohla být zmenšena až o 50 %. Kvůli tomu jsou výsledné soubory větší a je tak dosaženo horších kompresních poměrů. V testech tedy implementace nijak nevynikala, ale také se nedá říct, že by zaostávala. Naopak kde se potvrdila silná stránka implementace, bylo kódování velmi častých jevů, což jsou například symboly s velkou pravděpodobností výskytu. Implementace je v dobrém stavu pro další optimalizace ať už na úrovni algoritmu, nebo programovacího jazyka.

6 Závěr

Tato práce byla zaměřena na entropické kódování, především pak na metodu Asymmetric Numeral Systems (ANS). Z počátku byla představena komprese včetně jejího dělení a zmínění známých zástupců jednotlivých skupin. Blíže byli představeni nejznámější zástupci entropického kódování, kterými jsou Huffmanovo a aritmetické kódování. Byl popsán základní princip metody ANS a rozdělení na další varianty. Podrobněji byla rozebrána varianta s tvorbou konečného automatu, tedy tANS. Byly popsány důležité vlastnosti, kódovací a dekódovací postup. Postup tANS byl rovněž porovnán s Huffmanovým a aritmetickým kódováním. Závěrem této kapitoly obsahuje pohled na reálné implementace metod ANS a jejich využití.

Hlavní část práce se věnovala implementaci algoritmu Finite State Entropy (FSE), který je reálnou implementací metody tANS. Popisem důležitých funkcí a jiných částí této implementace se zabývala kapitola 4. Dle výsledů pozorování byla implementace tohoto algoritmu úspěšná a vznikl tak konzolový program, který je schopen zkomprimovat nebo dekomprimovat vstupní soubor. Při spuštění programu lze parametrem zadat počet stavů konečného automatu.

V další části následovalo testování vlastní implementace na vybrané kolekci souborů. Nejprve byly všechny soubory zkomprimovány a dekomprimovány se základním nastavením pro získání přehledu a výchozího bodu pro porovnání s dalšími testy. Jeden z testů byl zaměřen na určení optimální velikosti konečného automatu, kde se ukázalo, že optimální velikosti nelze jednoznačně určit pro všechny soubory, jelikož pro každý je ideální hodnota jiná. Velikostí automatu se zabýval ještě další test, tentokrát ovšem z pohledu rychlosti komprese a dekomprese. Pokud spojíme výsledky těchto dvou testů, lze konstatovat, že výchozí zvolená hodnota pro velikost automatu (4096) je vhodná z důvodu dosažení přijatelné míry komprese bez velkého vlivu na rychlost algoritmu. Testován byl také vliv rozložení symbolů na účinnost komprese. Výsledky tohoto testu potvrdily, že čím optimálněji jsou symboly rozloženy, tím lepší komprese lze dosáhnout. Nicméně zlepšení nebylo tak velké, jak se původně očekávalo. Při porovnání výsledků s Huffmanovým a aritmetickým kódováním měla implementace mírně horší výsledky, což je převážně způsobeno nedostatečnou optimalizací dekódovací tabulky, která je zbytečně velká. Obecně byly výsledky testů uspokojující a většinou potvrdily počáteční očekávání.

Vlastní implementace je v dobrém stavu pro budoucí vylepšení, především již zmíněnou optimalizací dekódovací tabulky. Dalším zlepšením by mohla být snaha nalézt optimálnější rozložení symbolů bez velkého vlivu na výslednou rychlost. Poté už by mohlo následovat zjednodušení programových konstrukcí za účelem optimalizace procesorových instrukcí. Větší změnou je spojení s některou slovníkovou metodou po vzoru programů Zstd nebo LZFSE. Tím by se mělo dosáhnout obecně lepších výsledků pro různorodé typy vstupních dat. Tento přístup se ukázal jako velmi efektivní a pomalu nahrazuje dříve nejpoužívanější metody pro kompresi dat.

Literatura

[1] SHANNON, C. E. *A Mathematical Theory of Communication*. [online]. [cit 2019-06-26]. dostupné z: http://math.harvard.edu/~ctm/home/text/others/shannon/entropy/entropy.pdf

[2] HUFFMAN, David A. *A Method for the Construction ofMinimum-Redundancy Codes*. [online]. [cit 2019-06-26]. dostupné z: http://compression.ru/download/articles/huff/huffman_1952_minimum-redundancy-codes.pdf

[3] DUDA, Jarek. *Asymmetric numeral systems*. [online]. [cit 2019-06-26]. dostupné z: https://arxiv.org/abs/0902.0271

[4] LEEUWEN, J. van. *On the construction of Huffman trees*. [online]. [cit 2019-06-26]. dostupné z: http://www.staff.science.uu.nl/~leeuw112/huffman.pdf

[5] RISSANEN, J. J. *Generalized Kraft Inequality and Arithmetic Coding*. [online]. [cit 2019-06-26]. dostupné z: https://pdfs.semanticscholar.org/9fcb/8d85e3d429f3816861fc7999e1bb68eefd39.pdf

[6] RISSANEN, J. J. – LANGDON, G. G., Jr. [online]. [cit 2019-06-26]. dostupné z: https://web.archive.org/web/20070928023306/http://researchweb.watson.ibm.com/journal/rd/232/ibmrd2302G.pdf

[7] MAHONEY, Matt. *Data Compression Programs*. [online]. [cit 2019-06-26]. dostupné z: http://mattmahoney.net/dc

[8] DUDA, Jarek. *Asymmetric numeral systems:entropy coding combining speed of Huffman codingwith compression rate of arithmetic coding*. [online]. [cit 2019-06-26]. dostupné z: https://arxiv.org/pdf/1311.2540.pdf

[9] GIESEN, Fabian "ryg". *rANS in practice*. [online]. [cit 2019-06-26]. dostupné z: https://fgiesen.wordpress.com/2015/12/21/rans-in-practice

[10] COLLET, Yann. *Finite State Entropy - A new breed of entropy coder*. [online]. [cit 2019-06-26]. dostupné z: http://fastcompression.blogspot.com/2013/12/finite-state-entropy--new-breed-of.html

[11] COLLET, Yann. *FSE decoding : how it works*. [online]. [cit 2019-06-26]. dostupné z: http://fastcompression.blogspot.com/2014/01/fse-decoding-how-it-works.html

[12] DUDA, Jarek. *Asymmetric Numeral Systems*. [online]. [cit 2019-06-26]. dostupné z: https://www.dropbox.com/s/f13ylst1mghrje5/ANSsem_mat.pdf

[13] COLLET, Yann. *FSE : Defining optimal subranges*. [online]. [cit 2019-06-26]. dostupné z: http://fastcompression.blogspot.com/2014/02/fse-defining-optimal-subranges.html

[14] BLOOM, Charles. *Understanding ANS – 12*. [online]. [cit 2019-06-26]. dostupné z: http://cbloomrants.blogspot.com/2014/02/02-18-14-understanding-ans-12.html

[15] BLOOM, Charles. *Understanding ANS – 8*. [online]. [cit 2019-06-26]. dostupné z: http://cbloomrants.blogspot.com/2014/02/02-06-14-understanding-ans-8.html

[16] COLLET, Yann. *FSE : distributing symbol values*. [online]. [cit 2019-06-26]. dostupné z: http://fastcompression.blogspot.com/2014/02/fse-distributing-symbol-values.html

[17] COLLET, Yann. *FSE encoding : how it works*. [online]. [cit 2019-06-26]. dostupné z: http://fastcompression.blogspot.com/2014/02/fse-encoding-how-it-works.html

[18] COLLET, Yann. *FSE encoding, Part 2*. [online]. [cit 2019-06-26]. dostupné z: http://fastcompression.blogspot.com/2014/02/fse-encoding-part-2.html

[19] COLLET, Yann. *FSE encoding : Mapping sub-ranges in a memory efficient way*. [online]. [cit 2019-06-26]. dostupné z: http://fastcompression.blogspot.com/2014/02/fse-tricks--memory-efficient-subrange.html

[20] COLLET, Yann. *Huffman, a comparison with FSE*. [online]. [cit 2019-06-26]. dostupné z: http://fastcompression.blogspot.com/2014/01/huffman-comparison-with-fse.html

[21] COLLET, Yann. *A comparison of Arithmetic Encoding with FSE*. [online]. [cit 2019-06-26]. dostupné z: http://fastcompression.blogspot.com/2014/02/a-comparison-of-arithmetic-encoding.html

[22] COLLET, Yann. – HANDTE, Felix – TERRELL, Nick. *5 ways Facebook improved compression at scale with Zstandard*. [online]. [cit 2019-06-26]. dostupné z: https://code.fb.com/core-data/zstandard

[23] COLLET, Yann. *Cyan4973/FiniteStateEntropy*. [online]. [cit 2019-06-26]. dostupné z: https://github.com/Cyan4973/FiniteStateEntropy

[24] *Apple Open-Sources its New Compression Algorithm LZFSE*. [online]. [cit 2019-06-26]. dostupné z: https://www.infoq.com/news/2016/07/apple-lzfse-lossless-opensource

[25] *CRAM benchmarking*. [online]. [cit 2019-06-26]. dostupné z: http://www.htslib.org/benchmarks/CRAM.html

[26] BLOOM, Charles. *Understanding ANS – 10*. [online]. [cit 2019-06-26]. dostupné z: http://cbloomrants.blogspot.com/2014/02/02-11-14-understanding-ans-10.html

[27] BLOOM, Charles. *Understanding ANS – 11*. [online]. [cit 2019-06-26]. dostupné z: http://cbloomrants.blogspot.com/2014/02/02-14-14-understanding-ans-11.html

[28] *The Canterbury Corpus*. [online]. [cit 2019-06-26]. dostupné z: http://corpus.canterbury.ac.nz/descriptions/#large

[29] MAHONEY, Matt. *About the Test Data*. [online]. [cit 2019-06-26]. dostupné z: http://mattmahoney.net/dc/textdata.html

[30] *Static Huffman Encoder*. [online]. [cit 2019-06-26]. dostupné z: https://sourceforge.net/projects/statichuffmanencoder

[31] *64-bit Range Coding and Arithmetic Coding*. [online]. [cit 2019-06-26]. dostupné z: https://sachingarg.com/compression/entropy_coding/64bit/

Přílohy

Příloha A – Implementace algoritmu FSE

Příloha v IS EDISON. Zdrojový kód vlastní implementace algoritmu FSE včetně části testovacích souborů.